Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey Campus

Doctoral Program in Artificial Intelligence

A Multiagent Approach to Outbound Intrusion Detection

Ph.D. thesis

by

Salvador Mandujano

December – 2004

Sophos Ex. 1007
Sophos Ltd. and Sophos Inc., v. Webroot Inc.
IPR2023-00731
Page 1 of 216

A Multiagent Approach to Outbound Intrusion Detection

A dissertation submitted by

Salvador Mandujano

to the faculty of the

Graduate Program in Electronics, Computing, Information and Communications

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Artificial Intelligence

Instituto Tecnológico y de Estudios Superiores de Monterrey

Monterrey, Mexico **December – 2004**

Sophos Ex. 1007
Page 3 of 216

Copyright © Salvador Mandujano, 2004

Contents

List of Figures **xi**

List of Tables **xvii**

1 Introduction **3**

1.1 Context and basic concepts 3

1.1.1 Information security fundamentals 3

1.1.2 The critical need for vigilance 5

1.2 Limitations of intrusion detection systems 6

1.3 Solution approach 7

1.4 Thesis statement 8

1.5 Document organization 9

2 Background **11**

2.1 Intrusion detection 11

2.1.1 Intrusion detection systems 11

2.1.2 Classification of intrusion detection systems 13

2.1.3 Detection techniques 15

2.1.4 Data collection and fusion 27

2.1.5 Challenges of current models 30

2.2 Recent trends in information security 33

2.3 Connection back-tracing 37

2.3.1 Limited troubleshooting visibility 37

2.3.2 Related projects 37

2.4 Agent-based intrusion detection 38

2.4.1 Agents 39

Sophos Ex. 1007
Page 7 of 216

2.4.2 Multiagent systems . . . 40
2.4.3 Multiagent components of intrusion detection architectures . . . 41
2.5 Taxonomies and ontologies in information security . . . 43

3 Problem description 49
3.1 Outbound intrusion detection . . . 49
3.1.1 Objective . . . 49
3.1.2 Main characteristics . . . 50

4 A multiagent outbound intrusion detection architecture 59
4.1 A multiagent approach to intrusion detection . . . 59
4.2 Specific objective of the architecture . . . 60
4.3 Characterization of automated attack tools . . . 60
4.3.1 Attack program signatures . . . 61
4.4 Environment . . . 63
4.5 Task definitions . . . 64
4.6 Task-to-agent mapping . . . 66
4.7 Organization of single agents . . . 68
4.8 Agent cells . . . 71
4.8.1 Agent classes and cell classes . . . 72
4.8.2 Signature and ontology deployment . . . 73
4.8.3 Operational outline . . . 74
4.9 Ontological model . . . 75
4.9.1 Supporting agent-based security technologies with ontologies . . . 76
4.9.2 Target-centric ontology for intrusion detection . . . 77
4.10 Source-centric ontology for outbound intrusion detection . . . 82
4.10.1 Use of the ontology . . . 86
4.11 Session tracing . . . 88
4.11.1 Internal design . . . 88
4.11.2 Hop identification . . . 89

5 Implementation 91
5.1 Technologies . . . 91
5.2 Triggering event . . . 92
5.3 Agent creation . . . 93

5.4 Network data capture . . . 94
5.4.1 Buffering and coordination . . . 95
5.4.2 Packet read backup . . . 96
5.5 Process data capture . . . 97
5.6 Matching of network signatures . . . 98
5.6.1 Primary data structure . . . 100
5.6.2 Splitting power analysis . . . 102
5.6.3 Analysis of network data . . . 103
5.6.4 Matching method for network signatures . . . 106
5.7 Matching process signatures . . . 108
5.7.1 Primary data structure . . . 109
5.7.2 Matching method for process signatures . . . 110
5.7.3 Signature fusion . . . 111
5.8 Session tracing implementation . . . 113
5.9 Reaction methods . . . 113

6 Experiments and results 115
6.1 Experimental objectives . . . 115
6.2 Test data . . . 115
6.2.1 Exploit types . . . 115
6.2.2 Target platforms . . . 116
6.2.3 Programming languages . . . 117
6.2.4 Test program set . . . 117
6.3 Experimental setup . . . 119
6.4 Signature generation . . . 120
6.5 A word on compiled and interpreted code . . . 120
6.6 Experiments: Signature matching and correlation . . . 120
6.6.1 Traffic generation . . . 121
6.6.2 Workload control . . . 122
6.6.3 Cell accuracy . . . 123
6.6.4 Cell performance . . . 127
6.6.5 Correlation accuracy . . . 130
6.6.6 Correlation performance . . . 131
6.7 Interactive session tracing . . . 132

6.8 Results 133

7 Conclusions, contributions and future work 135

7.1 Conclusions 135

7.2 Contributions 137

7.3 Future directions 138

Appendix A YAWL symbols A–141

Appendix B TCP and IP datagram formats B–143

Appendix C Outbound intrusion detection ontology in OWL C–149

Appendix D Classification Trees D–155

Appendix E Process signatures E–165

Appendix F Network signatures F–171

List of Figures

2.1 Anomaly-based intrusion detection. 13

2.2 Misuse-based intrusion detection. 15

2.3 A frequency system-call profile describing the behavior of a program [147]. 16

2.4 A simple Bayesian network connecting intrusion-related variables. CPT's store a-priori probabilities such as the probability of intrusion and the proportion of disk fragmentation 17

2.5 A sample neural network with window that predicts the next command once n commands have appeared in the input (window size is $n = 3$. 19

2.6 A Petri net corresponding to a partial order of events leading to a compromise state [103] 20

2.7 A parse tree corresponding to a program that can be evolved through Genetic Programming to detect a security-relevant event [41]. 21

2.8 Code vulnerable to buffer overflow before and after embedding a sensor into it. On the left side, lines 2–6 form the embedded sensor [223]. 22

2.9 Caller-callee relationships are distinctive of a program and can be used to identify abnormal behavior. An unexpected transition may be indicative of anomalous activity (see dashed line; no such transition exists in the actual program). 23

2.10 SVM splitting the feature space into two classes (they could represent normal and irregular behavior). 24

2.11 Example of DNA sequence alignment [39]. 26

2.12 The increasing sophistication of attacks and the decreasing expertise of users [2, 126]. 34

2.13 Number of security incident and number of vulnerabilities reported per year to the CERT–CC (*Coordination Center, Computer Emergency Response Team* [32]). 34

Sophos Ex. 1007
Page 11 of 216

2.14 Use of information security technologies (*2003 Global Security Survey, Deloitte & Touche* [48]) and intrusion detection technologies (*CSI/FBI Computer Crime & Security Survey 2003* [169]). The charts show the percentage of surveyed companies that report using this type of software. 35
2.15 Number of known Denial of Service (DoS) vulnerabilities and attacks per year (*PestPatrol's detection database, PestPatrol Inc. 2002* [155]). 36
2.16 (a) Replication is performed by a pool of agents which are contacted through a proxy agent. (b) Replication can also be peer-to-peer. 39
2.17 Hierarchical component of a multiagent intrusion detection architecture [224]. . . 42
2.18 Enabling high-level agent functions through an ontology layer. 42

3.1 Data collection focus of different intrusion detection approaches. 50
3.2 Attacks are detected at the originating host or at an intermediate host and can be blocked. This creates a security structure centered on the protection of others. 51
3.3 Characterization of automated attack tools through the use of locally-measurable and observable variables. 53
3.4 Under the assumption that the number of outgoing network packets an internal host generates is less than the incoming network packets a gateway sees, OID allows for more efficient packet processing. 54
3.5 A mesh of trusted OID agents that share plaintext of encrypted attack data over secure channels 56
3.6 Performance of an OID system: *Number of comparisons* vs. *time* (a comparison tries to match program files against program signatures). 57

4.1 Features of the OID environment 64
4.2 Primary OID tasks (YAWL diagram; see Appendix A). 65
4.3 Agent interconnection architectures (S = sensor, C = correlator, R = reactor; arrows indicate data exchange flow). 69
4.4 Grouping agents functionally into specific-purpose sub-environments or *agent cells*. 71
4.5 OID architecture and main cell classes. 72
4.6 Internal dataflow: (a) Traffic-signature cells, and (b) Process-signature cells (see Appendix A for YAWL symbols). 74
4.7 Internal dataflow: (a) Traffic-monitoring cells, and (b) Process-monitoring cells (see Appendix A for YAWL symbols). 75
4.8 Dataflow inside Correlation cells (see Appendix A for YAWL symbols). 76

4.9 Deployment of attack program signatures. Repositories connected to the Internet receive signatures from registered signature generation sites. Agents download signatures and ontology updates (M represents a mirror repository). 76
4.10 Graphical notation 77
4.11 A target-centric ontology for intrusion detection [203, 202]; bold arrows represent subclasses; soft arrows represent properties. 79
4.12 A source-centric ontology for outbound intrusion detection (see Appendix B for its definition in OWL) 80
4.13 Overlapping between the target-centric intrusion detection ontology and the source-centric OID ontology. 81
4.14 OWL statements defining the Signature–Process Signature subclass predicate. . . 81
4.15 Literal attributes of the *Traffic Signature* object. 81
4.16 OWL definition of a sample Traffic Signature instance. 82
4.17 The Agent-cell class 83
4.18 The Agent class 83
4.19 The Data-source class 84
4.20 The Signature class 85
4.21 The Reaction class 85
4.22 The Process class 85
4.23 The Repository class 86
4.24 The IP-packet class 86
4.25 The Executable class 87
4.26 Loop unwinding. If a user connects from host h_s to host h_t doing a loop through hops 1, 2, 3, 4, 5 and 6 as shown above, agent A at h_t will start the tracing process and will be able to identify the loop to reduce tracing time in the future. 90

5.1 Sample TCP socket table. 93
5.2 Hierarchy of classes derived from the primary JADE-agent class. 94
5.3 Packet reading assignment for two ($n = 2$) monitoring cells. 95
5.4 Failure cascade for sensors performing packet capture. 96
5.5 Sample backup structure for $m = 3$: when A is a primary agent, B is its first backup and C is its second backup; when B is a primary agent, C is its first backup and A is its second backup; when C is a primary agent, A is its first backup and B its its second backup. 97

5.6 Payload data of a `telnet` packet. One byte is used for sub-protocol window size negotiation. The maximum value corresponds to FF which produces a deviation between 0 and 30 in the hash value. 99
5.7 Network-traffic signatures of three different attack programs considering executions of *only one variant* per program. 99
5.8 Network-traffic signatures of three different attack programs considering *two variants*. 100
5.9 Snort's signature matching linked list [170, 37]. 101
5.10 Data structure for attack signatures. Nodes are sorted from highest to lowest frequency. 102
5.11 Splitting power per signature attribute. 104
5.12 The matching method gives more weight to attributes with high splitting power (see Table 5.4). 107
5.13 Function to compute payload hash σ_i sub-scores. 107
5.14 The text segment can be used to characterize a program. The figure shows an excerpt from the text segment of `smurf4`, one of the surveyed attack tools. 109
5.15 Process signature linked list. Every s_i node represents a program-variant signature. 110
5.16 Fusing hypothesis H_1 and H_2 into H_3. 112

6.1 Sample attack program containing root-shell code for two different target platforms: Linux and FreeBSD. 116
6.2 Experimental setup for the testing phase. 119
6.3 The Mucus intrusion detection stimulator. 121
6.4 Average hypothesis score $\sigma(x)$ corresponding to each attack tool using one traffic-monitoring cell. 123
6.5 Average hypothesis score $\sigma(x)$ corresponding to each attack tool using two traffic-monitoring cells. 124
6.6 Average hypothesis score $\sigma(x)$ corresponding to each attack tool using three traffic-monitoring cells. 125
6.7 Average hypothesis score $\pi(x)$ corresponding to each attack tool using one process-monitoring cell. 126
6.8 Average hypothesis score $\pi(x)$ corresponding to each attack tool using two process-monitoring cells. 127

6.9 Detection time with network monitoring under different traffic levels using one, two, and three cells. 128
6.10 Detection time with process monitoring under different workload levels using one and two cells. 129
6.11 Detection accuracy using correlation with one network-monitoring cell, one process-monitoring cell, and one correlation cell. 129
6.12 Detection accuracy using correlation with two network-monitoring cells, two process-monitoring cells, and two correlation cells. 131
6.13 Left: average detection time using two monitoring cells and one correlation cell. Right: average detection time using two pairs of monitoring cells and two correlation cells. 131

7.1 TCP datagram format (RFC 793) [38]. B–143
7.2 IP v.4 datagram format (RFC 791) [38]. B–145

List of Tables

2.1 Classification of intrusion detection systems by input type [223, 13]. 29
2.2 Classification of intrusion detection systems by their data analysis structure [223, 13]. 31
2.3 Classification of intrusion detection techniques 47

4.1 Description of the Sensor, Correlator, and Reactor agent classes 67
4.2 Description of the Name and Ontology Server agent class 68

5.2 Attributes stored on each node type. 104
5.4 Splitting power per attribute in network signatures. 106
5.6 Rules to implement automated response. 114

6.2 Automated remote-attack tools used in the experiments. 118
6.4 Matching scores for one, two, and three traffic-monitoring-cell scenarios. 125
6.5 Tracing time with and without encryption (RSA key exchange + DES chaining block ciphering); m corresponds to the number of connection hops or length of the chain. 132

Sophos Ex. 1007
Page 17 of 216

Abstract

"A Multiagent Approach to Outbound Intrusion Detection". Ph.D. dissertation by Salvador Mandujano, Instituto Tecnológico y de Estudios Superiores de Monterrey. Advisor: Prof. Arturo Galván. December – 2004.

This is a dissertation on the topic of intrusion detection. It supports the philosophy of system vigilance by exploring the concept of outbound intrusion detection, which is concerned with the identification and collection of evidence that helps prove local resources are being used to compromise external systems.

We discuss the motivation behind the approach, explain the need for splitting the scope of intrusion detection into sub-problems, and present trends in computer security that reveal basic design considerations that need to be taken into account when developing modern information security tools.

We propose a multiagent architecture for outbound intrusion detection supported by an ontology. Groups of agents collectively monitor outbound network traffic and local activity in order to identify references to neighboring systems that may be indicative of a compromise attempt.

We organize agents into sub-environments called agent cells that are connected to each other in a non-hierarchical fashion. Different classes of agents and cells compose the system, which performs attack modeling by employing multiple concurrent agents. Detection cells implement independent misuse intrusion strategies whose output is systematically fed to correlation cells capable of more accurate diagnosis.

We present an attack-source-centric ontology that extends previous work in the area. It enables message interpretation and enhanced agent communication within the architecture simplifying at the same time system maintenance and facilitating the integration of new components.

We describe the implementation of the proposed architecture through the FROID prototype as a proof of concept. This is a misuse-based intrusion detection system built with agent and semantic web open-source technology whose particular focus is the identification of automated remote attack tools. It performs signature generation, matching, and correlation, and supports a signature deployment mechanism over the Internet.

We introduce a similarity matching method that improves the performance of existing algorithms by leveraging entropy and frequency properties of the input hereby reducing search time.

We link detection with incident response by procuring low false alarm rates that allow us to study local and external reaction methods to outbound intrusion events. We also present a component of the architecture that performs tracing of interactive sessions as a way of identifying the root location of a security event.

We describe the experimental design and report the results obtained with the prototype that show the feasibility of the approach as an alternate way of containing the impact of security incidents through the integration of a mesh of monitoring agents.

Chapter 1

Introduction

Existing computer security tools are focused on the protection of local resources. A different, unselfish approach would be to protect a system by making sure that others are not able to launch attacks toward it. By monitoring local activity and outbound traffic, a system should be able to identify compromise attempts originated locally so that they are immediately stopped. This dissertation explores this idea. The design and implementation of a prototype architecture serves as a proof-of-concept that also provides experimental results regarding the use of agent and semantic web technology in the development of security software. This first chapter describes the overall aim and structure of the dissertation.

1.1 Context and basic concepts

This section presents basic concepts pertaining the area of computer security and discusses the need for having monitoring systems that complement secure software design and systems maintenance practices in the task of guaranteeing the security of a computing platform.

1.1.1 Information security fundamentals

Information security has been a very active research area over the last two decades. High connectivity and massive access to network resources combined with the widespread use of vulnerable software has resulted in hundreds of security breaches that leave considerable losses behind [169] putting at risk computer systems used to conduct the electronic operations of corporations, governments, and individuals worldwide [126, 124].

Security is a complex term involving technical, social, and economic aspects. For the purposes of this dissertation, which concerns the security of computer systems, the concept of information security or simply **security**, can be summarized as follows:

DEFINITION 1

Security (1). The ability a system has to prevent failure and to achieve its design objectives in spite of failure – either it be an intentionally induced fault or a fault caused by an operational error.

Sophos Ex. 1007
Page 21 of 216

A secure system should behave as it is supposed to while resisting attempts to trick it into performing actions that are not part of its expected functioning [18].

The objectives that information security pursues can be condensed in five main goals that help understand the specific purpose of existing security software [179, 156]:

1. *Confidentiality.* To protect information from unauthorized disclosure (e.g., using encryption).
2. *Integrity.* To guarantee that an information resource is not unexpectedly modified so that it remains trustworthy (e.g., protecting documents using secure hash algorithms).
3. *Availability.* To ensure that a resource is accessible at the time it is required (e.g., implementing controlled-redundancy strategies).
4. *Authentication.* To effectively validate the identity of a party (e.g., using password-based access control).
5. *Non-repudiation.* To guarantee that no party is able to deny actions it actually performed (e.g., implementing digital signatures).

From these five points we can derive a functional definition of computer security that some experts and practitioners in the area use:

DEFINITION 2

Security (2). A concept that integrates the confidentiality, integrity, and availability of a system as well as its ability to guarantee authentication and non-repudiation to all users and applications.

Each of these aspects has different priority levels depending on the protected resource. The value of a resource determines what aspects should be protected and how much it is worth investing in its protection. A database table, for instance, may contain strategic information and sufficient resources should be allocated for its protection at all times. Preserving the confidentiality of a message, on the other hand, may be necessary for just a short period of time after which the message becomes public and it is no longer necessary to keep it private, or it may not matter who reads it as long as it remains unchanged. These guidelines or rules that describe the set of acceptable actions are the components of a **security policy** [156, 126, 18].

DEFINITION 3

Security policy. Set of guidelines that indicate what and how resources should be protected by explicitly specifying all the valid ways in which the components of a system are allowed to interact.

Security policies are a basic construct in information security. They are used to differentiate abnormal from normal behavior. Since software is used to handle and store electronic information, it should be designed as to guarantee compliance with the security policies at the site and must be robust enough as to resist attack attempts.

The growth of the Internet has multiplied the number of inexpert users. Many of them are not familiar with even the more basic concepts of security and are often in charge of computer

systems and data banks [2, 4]. As a result, security policies do not always exist and the way users try to protect their systems is rather empirical, insufficient, and incorrect [18]. Living in a market-driven economy where often what matters is to deliver a product by the deadline regardless of quality, security is not always a priority for software firms. They overlook this aspect and release products that may be very good from the functionality point of view, but that are weak at the security level. If these products are not properly configured and maintained, they will become vulnerable systems that could be the target of attacks or be infected by viruses. In either case, the productivity of a company could be impacted [169, 124].

Computer security is a real problem and also a very fertile research area in computer science. New threats appear every day challenging the quality and configuration of all kinds of platforms ranging from web farms to even wireless telephony. Viruses and worms such as Sasser, Nimda, Code Red, Blaster, and Slammer [225, 154] have hit thousands of computers resulting, at the very least, in millions of people losing a considerable number of effective work-hours. When a popular website is compromised by hackers or when a denial of service attack is underway, the media will not fail to report the incident [111, 84]. In cases like these, the reputation of entire institutions may be hurt and the consequences may cause a considerable loss of market share.

A system that has not been found to be vulnerable today could be brought down tomorrow. Software must be permanently maintained with patches and updates, which constitutes the most effective procedure for systematically enhancing the security of a computer system [60, 180]. This, however, does not suffice to counter the effects of new and existing security threats [2].

1.1.2 The critical need for vigilance

Software is developed to meet design objectives. Even though a program may cover all these objectives in an actual implementation, it may also reach unexpected states at execution time that escaped the mind of its designers [207, 60]. The conditions necessary for going into one of these states may seem so unlikely that systems analysts ignore them without explicitly indicating what the program must do in those special cases. If a user carefully modifies the code, execution environment, input, or configuration of a program, the conditions for running into one of those states may suddenly appear and the behavior of the system can become unpredictable and/or incorrect.

Undesirable results may reduce the usefulness of the program as well as the trust users have with the system as a whole [18]. Some of those unexpected states may actually allow users to access information they were not supposed to, or to execute code without being authorized. The security policy will be violated by these states compromising the confidentiality of data, the integrity of its resources, or the availability of its services.

Secure coding practices can be used to build more robust software, however, the possibility of human error is always present. Fundamental software such as operating systems, network protocols, and database management systems are complex creation of the human kind. This complexity extends to the applications that interact with them which represents a challenge to the construction of programs that meet their design criteria. As the number of components of a computer systems increases, programs grow larger requiring more lines of code to develop both basic and sophisticated tasks as well as to keep backward compatibility. Unfortunately, the likelihood of their containing programming errors increases as well [207, 163].

The existence of an exposure window between the discovery of a new **vulnerability** and the generation of a patch that fixes it, gives attackers the opportunity to infiltrate by taking advantage of security holes before they are removed [10, 163].

DEFINITION 4

Software vulnerability. A coding bug, configuration error, or design flaw that can be exploited to compromise the security of a system.

Additionally, system misconfiguration by unexperienced administrators is a problem that increases every year as the speed of growth of the Web outperforms the speed at which security education and training is being deployed [169]. Technology is developed to solve these issues but the effort seems not to suffice either. Even when firewalls, strong cryptography and other tools are available to protect the integrity and confidentiality of data, users keep sharing their passwords, insiders abuse their privileges, and backups are considered a cumbersome task by many system administrators [124, 126].

The imperfection of software and configuration management makes necessary the use of mechanisms that protect systems when their security fails. Before, during, and after the exposure window, systems must be guaranteed a protection level that limits the damage someone could cause by breaking into them. A number of solutions have been developed to help administrators deal with this issue. From among all solution strategies, system monitoring, also known as **vigilance**, has proved effective at controlling the impact of security incidents [4, 18].

Monitoring tools such as firewalls, anti-virus software, and **intrusion detection systems** scrutinize the activity of a system and constitute a protection layer that aims at identifying security breaches that escape the operating system and applications.

DEFINITION 5

Intrusion. Any violation attempt or actual violation to the security policy committed by an internal or external user [126].

DEFINITION 6

Intrusion detection. The set of techniques used to identify intrusions on computer systems. Intrusion detection is used to find attacks that cannot be prevented [60].

Assuming that code bugs, misconfiguration, privilege misuse, and lack of security training will continue to exist, it is critical to develop security vigilance solutions that complement good software engineering in order to protect the millions of computer systems connected to the Internet and used daily around the globe.

1.2 Limitations of intrusion detection systems

Intrusion detection technologies have been developed since the late 80's [49]. Many techniques have been implemented into commercial products and research prototypes following different

approaches. The results obtained from them have exposed clear opportunity areas that need to be considered in order to develop more effective and secure intrusion detection tools.

Among the most important challenges in the area, we can name the following four which are addressed in this dissertation:

- Low detection accuracy reflected by high rates of false positives and false negatives that limit the usefulness of this type of security software.
- Immense amounts of data that need to be inspected in order to detect signs of intrusive activity.
- High overhead generated by the amount of resources that need to be allocated by intrusion detection tools in order to achieve real-time operation.
- Reduced portability and lack of standards that impede intrusion detection systems to share information (e.g., attack signatures) and to be integrated with other technologies.
- Limited connection between intrusion detection and automated response.

Most intrusion detection systems suffer from these limitations [126, 2]. One of the most recent models proposed for solving the problem of intrusion detection has been the use of agent-based systems. These systems take advantage of the distributed nature of the problem and leverage some of the basic and most attractive features of software agents. These systems also have some common design problems that need to be taken into consideration should we want to develop robust agent-based information security solutions:

- Frequent use of hierarchical structures for the arrangement of agents which results in potential bottle-necks and single-point-of-failure flaws.
- Limited autonomy of the agents composing this type of systems.
- Negligible use of standard communication and knowledge management mechanisms that may allow intrusion detection systems to integrate with other technologies and to facilitate their own maintenance.
- Lack of prompt generation and deployment of attack signatures that help quickly identify cyber-attacks.
- Limited security of the intrusion detection systems themselves.

1.3 Solution approach

In this dissertation, the goal of intrusion detection is complemented with an additional monitoring layer that concentrates on aspects rarely considered by existing intrusion detection systems: the inspection of outbound network traffic and the identification of attack attempts originated locally to compromise other systems in the outside.

This approach is studied as a way of limiting the extension of the damage an intruder may cause and constitutes the foundation of a type of security tools whose objective is to protect neighboring systems by blocking intrusion attempts launched locally toward them and by alerting them about the fact that they are the target of an attack. Therefore, for the sake of its own security, a system could be interested in other hosts having such a monitoring system in place.

In particular, instead of modeling the intruder's behavior (i.e., human behavior) as the way of identifying abnormal activity, our solution approach focuses on the modeling of surveillance and attack tools (i.e., program behavior.) Attack software constitutes a repository of security information readily available on the Internet which has remained greatly unexplored and that could help improve the development of security tools. By detecting a rogue program in execution we expect to be able to protect other systems from being victims of a break-in. Further incidents could be prevented by countering the attack at the root or at an intermediate point instead of just assuming the defensive role of the target.

Given the nature of the problem, agent teams are used to implement this approach to intrusion detection. An evidence correlation mechanism is utilized to increase detection accuracy at a level that enables the integration of the detection task with automated response to security incidents.

The perspective of this dissertation regarding intrusion detection allows for the definition of a broader security context, poses new research questions, and provides an alternate way of approaching the security monitoring problem from a cooperative, less selfish stand-point.

1.4 Thesis statement

This dissertation has been developed to validate the following main hypotheses:

1. Preventing compromise attempts launched locally toward external targets is an effective way of limiting the effects of a security threat.

2. The data bank comprised of all the remote attack tools available on the Internet constitutes a valuable source of security information that can be used to develop better protection mechanisms.

3. The use of more than one software agent for the modeling of a rogue program produces more accurate detection results.

4. The generation and deployment of remote-attack program signatures or *fingerprints* can be automated.

5. A multiagent intrusion detection architecture supported by an ontological model requires less maintenance than non-agent software.

6. The use of virtual machines in the construction of trusted sub-environments helps agents more easily validate the integrity of the receiving (potentially malicious) host.

1.5 Document organization

This document is organized as follows. Chapter 1 has presented an overview of the topics central to this dissertation as well as basic terms, the thesis statement, and the specific security aspects to address. Chapter 2 reviews related projects and background subjects in depth. It describes detection techniques, outlines the multiagent paradigm, and presents recent trends in computer security that motivate some of the main ideas of this work. Chapter 3 introduces the concept of outbound intrusion detection emphasizing its differences with the traditional intrusion detection approach. Chapter 4 is a description of the proposed architecture. It contains the definition of the ontology that supports the model as well as the organization of its components and their functions. Chapter 5 explains the implementation of the architecture, justifies some important design ideas, and describes the algorithms used to perform signature generation and matching. Chapter 6 describes the experimental design, presents results, and discusses findings from the execution of the proof-of-concept prototype under different scenarios. Chapter 7 lists the conclusions of the dissertation, summarizes its contributions, and identifies possible directions for future research. All cited references and the appendices appear at the end of the document.

Chapter 2

Background

This chapter is divided into five main sections. They present basic concepts used and studied throughout subsequent chapters.

We first define some intrusion detection terminology. Computer security trends observed in recent surveys are presented to support the motivation behind this dissertation. The problem of connection tracing, which is closely related to intruder identification, is also introduced. Finally, the multiagent approach to intrusion detection is explained as well as some relevant projects in the area of security taxonomies and ontologies.

2.1 Intrusion detection

This section introduces intrusion detection concepts relevant to the objectives of this dissertation. It also describes the "state of the art" on detection techniques, shows recent trends in computer security and cyber-crime, and lists some of the primary research challenges in intrusion detection.

2.1.1 Intrusion detection systems

There are several ways to avoid being the target of an intrusion or to react to one [76]:

1. *Prevention.* To proactively reduce the likelihood of intrusive activity by influencing the root causes of the problem (e.g., to remove unnecessary services from a host).

2. *Preemption.* To actually counter-attack the source of the intrusion before it reaches its objective (e.g., launching a denial of service attack toward the presumed offender [208]).

3. *Deterrence.* To persuade the attacker to stop the attack (e.g., using system banners announcing the presence of the security tools installed on the system).

4. *Deflection.* To make the attacker think he has succeeded (e.g., using honey-pot boxes [193]).

5. *Use of countermeasures.* To react to the intrusion as it develops (e.g., applying patches and reconfiguring the system).

Sophos Ex. 1007
Page 29 of 216

6. *Detection.* To identify intrusive attempts and incidents in order to keep the corresponding authority informed (e.g., reporting anomalous URL strings received by the webserver).

Out of these possible reactions, intrusion detection, along with prevention, is the approach that has been explored the most [13, 4]. Twenty years ago, Anderson wrote seminal work in the area which increased the interest in computer security [6]. A few years later, the first intrusion detection model by Denning opened the field and eventually became central reference to multiple security architectures developed around the world [49].

Intrusion detection is concerned with the identification of activities that have been generated with the intention of compromising the security of local resources [4]. An attacker may be trying to gain access to a system from outside of the network, but it can also be the case that a legitimate internal user is abusing the access control mechanism in order to disclose sensitive data or to modify files without having been authorized to do so.

Intrusion detection software gathers and analyzes data from different sources within the system in order to identify activity that may be indicative of a compromise attempt [223]. An intrusion detection tool will alert the administrators who will have to take action in order to limit the damage and fix any security hole present in the system.

A related concept that needs to be mentioned is that of *intrusion tolerance* [51, 217].

DEFINITION 7

Intrusion tolerance. The ability of a system to continue working despite of intrusion [132]. Its main goals are: (1) to recognize the failure event, (2) to asses the extent of the damage, (3) to stop the event, and (4) to recover essential services during or after the incident. It is also often referred to as *survivability*.

This discipline comprises the sub-area of *fault tolerance* which studies ways to make a system resistant to failure (fault tolerance does not contemplate the case of induced failures or intrusions [210]).

DEFINITION 8

Fault tolerance. The ability to resist failures that are natural to the system (i.e., random operational faults [157]).

Intrusion tolerance also studies the problem of designing modular systems so that components can be easily isolated in the event they are compromised hereby allowing the rest of the system to keep providing all or part of its overall functionality [99, 12].

It is important to make the point that intrusion detection defers from intrusion tolerance and fault tolerance in that intrusion detection does not have to do with the design and implementation of robust systems that resist failure or attack. Intrusion detection's only goal is to identify violations to a local security policy and to report them via alert messages to the administrators [63].

2.1.2 Classification of intrusion detection systems

There are two primary methods to implement intrusion detection: (1) *anomaly detection* (also known as *profile-based intrusion detection* and *statistical intrusion detection*), and (2) *misuse detection* (also known as *signature* or *pattern-based detection*).

DEFINITION 9

Anomaly detection. An intrusion detection technique that creates a definition of "normality" and flags as intrusive any activity that deviates from it [126].

Anomaly detection. Anomaly detection assumes intrusions constitute irregular events. It creates behavior profiles by observing the "normal" activity of users or applications (Figure 2.1). Once these profiles have been built through training or using heuristics, any activity that diverges from them will be considered as potentially intrusive. False positives and false negatives may arise due to the threshold that divides normal from abnormal behavior.

A **false positive** is the classification of legitimate activity as intrusive, whereas a **false negative** is the classification of illicit activity as benign – the latter being a much more serious problem as it represents failure to identify hazardous events [103].

Anomaly-based intrusion detection systems are computationally expensive due to the techniques they employ. They typically require the construction and tuning of behavioral models (some of these systems are described in [152, 108, 83, 175, 109, 64]). Classifying what is "normal" from what is not is a difficult task that involves not only technical factors such as compliance with protocol definitions or the understanding of the internals of an application, but also aspects such as the expertise level of the users, their preferences, and the date and time. They also contribute to the complexity of identifying what is an acceptable variation of the system's security context and they are sometimes difficult to model as input for software systems [4].

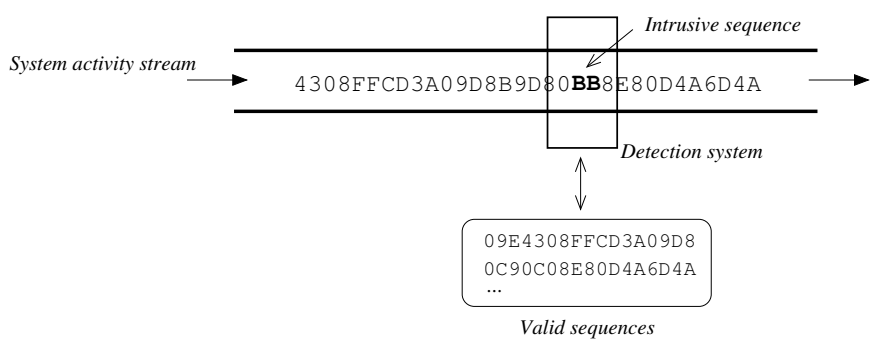


Figure 2.1: Anomaly-based intrusion detection.

– *Advantages of anomaly detection.* The main advantage of anomaly detection is that it can potentially discover attacks that have not been seen before [103]. Any evidence of activity variating from the profile will be considered for further analysis regardless of what has been previously seen. Another advantage is that the system can be left to run unattended once the

normality profile has been created. This considerably reduces maintenance time [60].

– *Disadvantages of anomaly detection.* One of the disadvantages of this approach is that unusual behavior is not always a sign of attack. This may produce false alarms that report incorrect information to the administrator if not implemented correctly [2]. Another problem is that anomaly detection systems could be tricked into accepting anomalous behavior: an attacker could make the system consider illicit activities as normal behavior if he has access to the system during the training phase [60]. In general, the methods to create and evaluate these profiles are computationally intensive and complicate real-time detection [2, 13].

DEFINITION 10

Misuse detection. An intrusion detection technique that defines patterns of abnormal activity and then looks for occurrences of those patterns in the monitored data streams. An occurrence signifies a potential intrusion [13, 126]).

Misuse detection. The misuse detection approach assumes that intrusive activity can be described through patterns and that those patterns can be captured by means of event strings or state machines [105, 187]. These systems identify *signatures* that contain information distinctive to an attack (Figure 2.2).

DEFINITION 11

Attack signature. A pattern or fingerprint that captures the essence of an attack and which can be used to identify the attack when it occurs [142].

Monitoring programs observe the different data streams present on the system and match them against a bank of attack signatures [170, 194, 53, 72]).

Misuse detection is a pattern matching method. Once signatures are generated, the problem reduces to one of search and matching. The primary difficulty of this approach is to generate signatures that correctly represent what constitutes intrusive actions [126]. Signatures that are too specific may not catch slight variations of an attack, and signatures that are too flexible might generate false alarms by flagging as intrusive activity that is not.

– *Advantages of misuse detection.* Misuse detection techniques do not consume as much resources as anomaly detection methods do. The processing of signatures can be highly optimized as to comply with performance expectations by adjusting parameters in the matching algorithms [194, 75]. Thus, the rate of false alarms can be controlled since matching accuracy and signature specificity constitute a trade-off directly bound to the security level these systems offer [126].

– *Disadvantages of misuse detection.* Signatures need to be very precise for them to be useful and the task of signature specification is rather time consuming and requires a high level of expertise [13, 126]. In the event an attack signature overlaps with licit behavior, false positives may arise. If the signature is not specific enough the attack will go unnoticed and will compromise the security of the system. The main disadvantage of this approach is that, as opposed to anomaly-based intrusion detection, it cannot detect threats for which a signature has not been produced. Signature banks must be permanently updated and signatures need to be defined and tuned in order to detect new attacks.

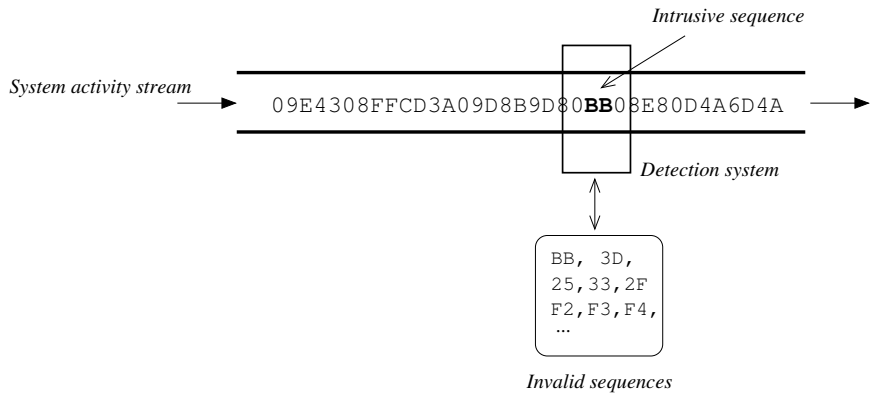


Figure 2.2: Misuse-based intrusion detection.

Hybrid methods. The two main approaches, anomaly and misuse detection, have advantages that complement each other. In fact, a breed of hybrid systems are the ones dominating the market and some more have become successful research projects (e.g., [164, 205, 77, 187, 46, 194, 140, 5].

These systems are fed with attack patterns they will detect and, at the same time, implement learning and profiling algorithms to necessary to identify invalid actions [90, 2]. The solution hybrid intrusion detection systems provide is richer detection-wise as they typically access two or more information sources which gives them the advantage of having more data elements to draw conclusions from. By correlating information of a normality profile with evidence of a particular attack, a system will be in a better position to accurately identify illicit behavior [60].

2.1.3 Detection techniques

In order to perform the actual task of detecting unusual events from evidential data, intrusion detection systems have made use of many different techniques. This section outlines the most representative methods developed to date (we include the techniques surveyed by Kumar and Spafford in [104] and extend the list with more recent detection mechanisms):

1. **Frequency-based detection**. This was one of the first anomaly detection methods developed (it first appeared in Denning's seminal paper [49]). It captures frequency patterns of users and programs through profiles that contain abnormality values $P_1, P_2, ..., P_n$ corresponding to a set of n system *measures* (i.e., representative system variables). In order to determine whether the system has encountered an unstable state, the observed frequency values f_i are combined through an expression such as:

$$\gamma = a_1 f_1^2 + a_2 f_2^2 + ... + a_n f_n^2; \quad a_i > 0 \qquad \ldots (1)$$

which captures frequency information for all the measures using a series of weights a_i. This technique detects intrusions by computing security level γ on a permanent basis and comparing it to a threshold value δ. If $\gamma > \delta$, an intrusion may be underway [182, 104].

The different measure types that can be evaluated are diverse and can be divided into three main types: (1) ordinal measures (such as the number of files open by a certain device in a given period of time), (2) distribution measures (such as the statistical distribution of CPU utilization during the day), and (3) categorical measures (such as the number of successful transfers and the number of failed transfers requested to a DNS server).

Profiles capturing normality values could be of the form:

```
<U, file, execute; Error(0), CPU(00:01-00:04), ExpectedExec(10:00-11:30)>
```

This could indicate that whenever user U executes $file$, no errors are expected, CPU processing is likely to fall between 1 and 4 seconds, and the execution is expected to take place between 10:00AM and 11:30AM. This way it is possible to identify an intrusion if it rises the value of a measure over the limits stipulated in the profile [66]. If a program is described through the system calls it uses, a frequency profile can be built in order to describe its most common behavior (Figure 2.3):

Sequence of system calls

```
...
read(6, "/dev/hda3 / ext3 rw 0, 4096) = 211
close(6)                                = 0
munmap(0x40017000, 4096)                = 0
open("/proc/stat", O_RDONLY)            = 6
fstat64(6, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
mmap2(NULL, 4096, MAP_PRIV|MAP_ANONY, -1, 0) = 0x40017000
read(6, "cpu  5989400 9772 1400819 982758"..., 4096) = 312
read(6, "", 4096)                       = 0
close(6)                                = 0
munmap(0x40017000, 4096)                = 0
open("/etc/toprc", O_RDONLY)            = -1 ENOENT
...
```

Frequency profile

close	9	connect	66
create	45	creat64	57
mmap2	38	dup	26
execve	27	fcntl	15
fdsync	68	fork	34
fstat	13	fstat64	17
read	74	getmsg	49
getgid	32	getpgid	68

Figure 2.3: A frequency system-call profile describing the behavior of a program [147].

This profile describes how often the program requests each of the system-call it uses. Taking this profile, if we observe that, for instance, the number of `close()` system-calls is up to 30, it is likely that there is a problem on the system.

2. **Probabilistic detection models**. Let $A_1, A_2, ..., A_n$ be n different measure variables which can take one of two values: 1 to indicate the measure is anomalous, and 0 otherwise. Let I be the hypothesis that the system is undergoing intrusion. The sensitivity of each measure A_i is determined by $P(A_i = 1|I)$ and $P(A_i = 1|\neg I)$. By combining these probabilities we can determine the likelihood that the system is undergoing attack given the values of the measures:

$$P(I|A_1, A_2, ..., A_n) = P(A_1, A_2, ..., A_n|I) \times \frac{P(I)}{P(A_1, A_2, ..., A_n)} \qquad ...(2)$$

This requires the joint probability distribution of the measures given I and $\neg I$ [104]. Assuming probabilistic independence of the measures, obtaining $P(A_1, A_2, ..., A_n|I)$ would be reduced to:

$$P(A_1, A_2, ..., A_n|I) = \prod_{i=1}^{n} P(A_i|I) \qquad \ldots (3)$$

and

$$P(A_1, A_2, ..., A_n|\neg I) = \prod_{i=1}^{n} P(A_i|\neg I) \qquad \ldots (4)$$

A more realistic example would take into account the interdependence of the various measures. Lunt et al. [115] accomplished this through the use of *covariance matrices*. The compound anomaly of the system using vector $A = [A_1, A_2, ..., A_n]$ is given by

$$A^T C^{-1} A \qquad \ldots (5)$$

where matrix C stores the interdependence between each pair of anomalies A_i and A_j at $C[i, j]$. Through a covariance matrix it is possible to consider the fact that the inputs that intervene in security diagnosis are interrelated [143].

3. **Bayesian networks**. Detection models using Bayesian learning take the basic structure of probabilistic detection in which a measure variable may affect others [183, 172]. This information is used to create belief networks called *Bayesian networks* in order to graphically represent causal dependencies between random variables. These networks are directed acyclic graphs and are able to compute joint distributions by specifying only a subset of probabilities corresponding to the neighbors of a node (Figure 2.4).

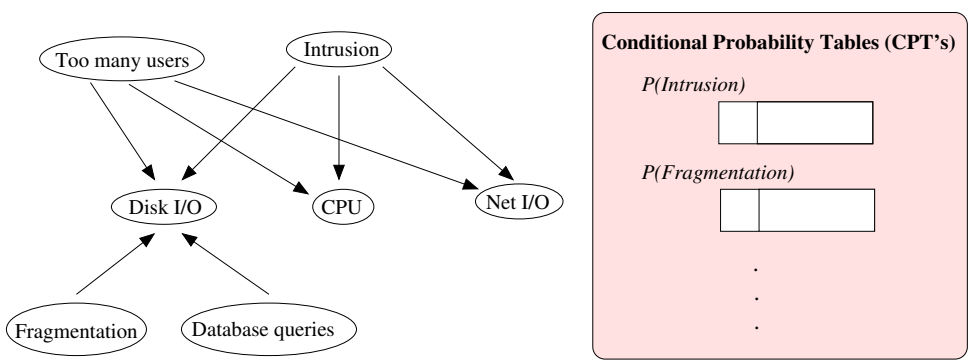


Figure 2.4: A simple Bayesian network connecting intrusion-related variables. CPT's store a-priori probabilities such as the probability of intrusion and the proportion of disk fragmentation

.

Bayesian networks use sets of *conditional probability tables* (CPT's) that store probability values corresponding to the random variables given certain preconditions. These probabilities are usually taken from the experience of past events and contribute to effectively predict the

desired variable [94]. Intrusion is determined with a certain probability given the presence or absence of evidence.

Figure 2.4 shows a small network which can be used to answer questions such as "what is the probability of intrusion given the observed level of disk fragmentation?", or "what is the probability of intrusion given that the number of current logged-on users is high and CPU load is low?"

4. **Conditional models using event sequences**. Instead of considering system measures, this is a technique that detects irregular activity based on sequences and sub-sequences of events $E = [e_1 \prec e_2 \prec ... \prec e_n]$, where e_i is an event of type i and $\prec$ represents event precedence in time. These algorithms try to come up with a value for $P(I|E)$ where I indicates the probability of intrusion and E the probability of occurrence of an event pattern. Bayes calculus [173] can also be applied to determine how likely intrusion is given that pattern E appeared:

$$P(I|E) = \frac{P(E|I) \times P(I)}{P(E)} \qquad \ldots (6)$$

An experienced system administrator could quantify approximates of a-priori probabilities such as the probability of occurrence of a given sequence of events at a certain time frame. It is then possible apply a probabilistic method that will be the one detecting intrusions and intrusion attempts. This model becomes complex when it is necessary to consider the large amount of intrusion patterns that can be observed on a system [104].

5. **Expert systems**. Expert systems have also been used in intrusion detection [181, 91, 105]. These systems embody the knowledge of an expert in order to identify unusual data and irregular actions. They are classified as anomaly-based as they explicitly specify the patterns to look for [103]. The success of these methods is directly related to two factors: (1) the expertise of the security officer that will be used as input to the detection mechanism, and (2) the effectiveness of the implementation to coherently structure the expertise of the human into a computer program.

   Expert systems have also been used to interpret system measures in intrusion detection, however, the well-known limitations of this approach for doing uncertainty reasoning have stopped the progress in the area [104].

6. **Inductive generalization**. This is a rule-based approach which characterizes the behavior of a system or user using inductive generalization under the assumption that system events can be predicted [80, 200]. A set of rules of the form:

$$E_a \rightarrow E_b \rightarrow E_c \Rightarrow (E_d = 0.95, E_e = 0.05) \qquad \ldots (7)$$

expresses the probability of seeing an event appear after a sequence of other events. For instance, once events a, b, and c have appeared sequentially, event d is 95% likely to appear, and e will appear with a probability of 5%. These are both normal, acceptable sequences.

Rules help to know beforehand the probability of being under attack given a set of observed patterns [200]. Events will be flagged as intrusive when they match the left-hand side of a rule but diverge on the probabilities that appear in right-hand side.

It is claimed that if an intruder tries to guide the rule-learning phase as to accept irregular activity as normal, the very meaning of rules can help determine something irregular has been captured [200] (i.e., rules are easily read and interpreted by administrators). This anomaly-based method has a major weakness: any unknown attack which does not match the left-hand side of a rule will not be even considered for further inspection.

7. **Neural networks**. Supervised learning methods have also been used in intrusion detection. These methods develop classifiers that predict output values based on a set of input attributes. Neural networks can be used for supervised learning and have proved effective at classifying different sorts of data when complemented with pruning, early stopping, and other heuristics [174, 27, 173]. A neural network contains a set of nodes organized into layers. Input and output layers are interconnected through one or more intermediate layers and the learning is performed by updating edge weights and adapting the parameters of the functions present on each node.

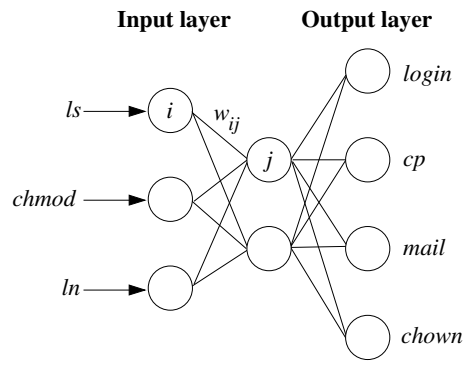


Figure 2.5: A sample neural network with window that predicts the next command once n commands have appeared in the input (window size is $n = 3$.

In the case of intrusion detection, a neural network can be used to to predict the next event to appear. For instance, the command history of a user can be used to train a neural network. Once training is done, a window of k commands will be sliding in time in order to monitor the activity of the user. If the neural network indicates that after certain k events, event e_j should follow and a different event is observed, an alarm is activated. Similarly, system calls can also be predicted after a training phase that will identify interdependencies among system-call streams hereby creating a sense of normality for the system [64].

Figure 2.5 shows a small network that predicts the next command given a three-command input. Command sequences must be divided into *windows* so that supervised learning can take place. In this case, the window will be moving one command at a time and will use three commands as input in order to determine which of the output values is more likely to

appear next. Training time is the main restriction of the neural-network approach for the problem of intrusion detection. However, it deals very well with noisy data [174].

8. **Attack scenario models**. This misuse-detection approach describes attacks through sequences of behaviors called *scenarios* [68]. A database containing attack scenarios is accessed by the monitoring system in order to tell which of the scenarios is most likely to be present at a given time.

   These systems update the database using evidential reasoning calculus. The use of a mathematically-sound theory like evidential reasoning makes retraction of intermediate conclusions easy as evidence to the contrary accumulates. When determining the behavior to look for next, the system tries to maximize a factor α_x defined by:

$$\alpha_x = \frac{P(S_x|B)}{P(S_x|\neg B)} \qquad \ldots (8)$$

   where S_x is a scenario and B corresponds to a behavior that confirms the presence of such scenario. Like other misuse detection systems, building attack scenario models require high levels of expertise and continuous updates [104].

9. **Petri nets**. This is a misuse detection technique which models attacks through Petri nets [105, 103]. Petri nets are graphs composed by states and conditions which have well defined semantics. The transitions from a start state into a final state describe the evolution of attacks. Petri nets have been used to model system events such as user commands and system calls (two popular types of input data used to detect intrusion).

   The graphical representation provided by a Petri net conveys an intuitive understanding of the events that compose an attack and allow for the representation of partial order as well – finite-state automata have also been explored in a similar way [201]. Rather than handle fixed sequences of events, Petri nets can describe series of events that are loosely connected to each other. Figure 2.6 shows a net describing an attack scenario. The steps that compose an attack may have variable precedence relations not being part of an absolute-order sequence.

```
1. cp /bin/sh /usr/spool/mail/root
2. chmod 4755 /usr/spool/mail/root
3. touch x
4. mail root < x
5. /usr/spool/mail/root
```

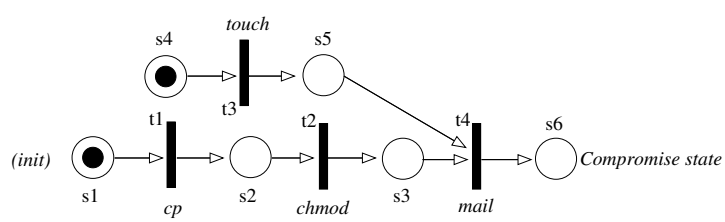


Figure 2.6: A Petri net corresponding to a partial order of events leading to a compromise state [103]

Petri nets allow us to describe and match partially ordered sequences as an alternative way of visualizing attack scenarios. In Figure 2.6, state $s5$ may be reached well before or after an

alternate transition reaches $s3$. "A partial order state transition analysis allows more than one sequence of events in the state transition diagram" [66]. A single diagram can be used to represent several intrusion scenarios involving the same set of actions. This simplifies modeling and speeds up detection.

10. **Genetic programming**. Many security systems use monolithic programs to perform intrusion detection functions. A different approach is to have multiple small programs in charge of simple monitoring tasks. These programs can be further evolved through Genetic Programming techniques so that the best candidate for a determined task can be found [41].

    Genetic programming is a machine-learning method derived from *Genetic Algorithms* which evolves a population of programs, and not a population of sample points, to solve a problem. Possible solutions are represented as parse trees whose component nodes are manipulated through genetic operations such as crossover and mutation. A fitness function aids in the identification of the best individuals (i.e., programs). In the case of intrusion detection, normal and abnormal activity is presented to the programs in order to evaluate how well they do at distinguishing between the two.

    Figure 2.7 depicts a sample parse tree corresponding to a block of code whose purpose is to watch for packets arriving at a host but destined for another host [41] (a simplistic way of determining something wrong may be happening).

```
FOR_EACH_PACKET DO
   IF( ip-destination-address-of-packet
          NOT_EQUAL my-ip-address )
     THEN Generate(alert-broadcast)
   END_IF
END_FOR
```

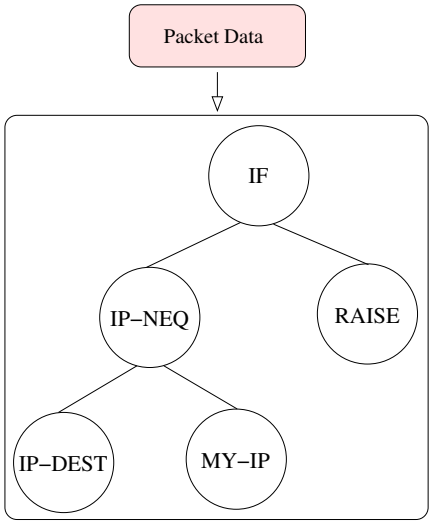


Figure 2.7: A parse tree corresponding to a program that can be evolved through Genetic Programming to detect a security-relevant event [41].

Each component of a program, which can be expressed using different languages and graph structures, will be moved from place to place connecting with other different nodes during the evolution stage. This will last until the component finds the location at which it can produce best detection results to the program as a whole. Important to notice that a good

number of combinations will produce incorrect trees from the programming logic perspective whose fitness will be low. These will be discarded and the best ones will be selected to implement the intrusion detection strategy.

11. **Embedded sensors**. Another technique called *embedded sensors* [223, 98] proposes the use of code segments that are inserted into programs during the development phase. These code segments automatically search for abnormal conditions at execution time including well-known error or attack conditions such as stack overrun and data-type abuse (Figure 2.8).

```
1 char buf[256];
2 strcpy(buf, getenv("HOME"));
```

(a) Code before inserting sensor

```
1 char buf[256];
2 {
3    if( strlen(getenv("HOME")) > 255) {
4       log_alert("buffer overflow");
5    }
6 }
7 strcpy(buf, getenv("HOME"));
```

(b) Code after inserting sensor

Figure 2.8: Code vulnerable to buffer overflow before and after embedding a sensor into it. On the left side, lines 2–6 form the embedded sensor [223].

The type of checking these tools provide is equivalent to the implementation of good coding practices with the difference that sensors can be specific to certain attack types and are reusable. Among the disadvantages of this approach are the need to have the source code of the program to protect, the need to implement sensors in the same language as the program, and reduced portability [59].

12. **Functional dependency checking**. This is a recent misuse detection approach based on functional dependencies present in a program. In order to perform certain actions, a function executes others that provide the results it needs. At each function call, one function is the *caller* and other is the *callee*. Caller-callee relations can be captured through a graph using executable or source code: at run time, the runtime stack can be read in order to determine what are the functions being called by each function; by looking at the source code, it can be determined the same type of information but at a higher level [71].

    These dependencies are characteristic to a software program and it is possible to implement a monitor which detects graph transitions not present in the graph (system-call precedence as well as regular function-calls can be used; [211, 186, 201]).

    Figure 2.9 shows an example of this approach. The structure of a program is analyzed as to create a graph describing the relationships among its functions. This graph is built during a training phase when legitimate behavior is exercised and captured by the graph. Once in detection mode, if a security monitor observes a transition that is not present in the program's graph, it is likely that the program has been altered [186].

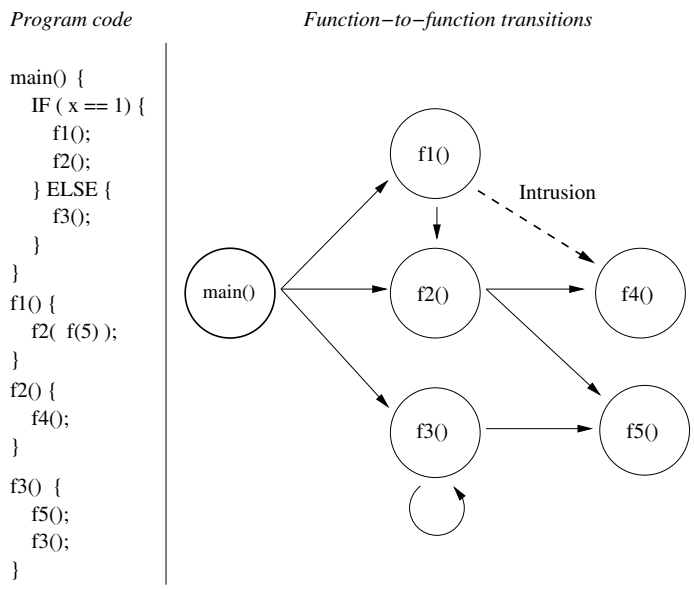


Figure 2.9: Caller-callee relationships are distinctive of a program and can be used to identify abnormal behavior. An unexpected transition may be indicative of anomalous activity (see dashed line; no such transition exists in the actual program).

13. **Data mining**. Some researchers have studied ways to apply data mining techniques to intrusion detection [114, 113]. Data mining is concerned with the discovery of new knowledge in large data collections. This discipline employs machine learning and statistical analysis methods to identify hidden patterns and trends a human would not be able to detect efficiently on large databases. Inferred patterns or rules allow the prediction of future results and, in the case of intrusion detection where the amount of data to inspect is quite large, this kind of prediction can help detect furtive attempts to circumvent the security of a system.

    For example, suppose that 10% of the packets received by a host have the SYN and ACK flags on and that 30% of the packets that have the ACK flag on also have the SYN flag on. These two variables could be associated using a rule $ACK \Rightarrow SYN$ with *support degree* $s = 0.1$ and *rule confidence* $c = 0.3$:

    $$ACK \Rightarrow SYN \qquad s = 0.1, c = 0.3 \qquad \ldots (9)$$

    Rules like this can be computed for different components in order to build a normality profile that helps identify illicit activities [112].

14. **Fuzzy data mining** has also been studied [22]. The integration of fuzzy logic with data mining helps create more abstract patterns at a higher level. Rather than describing a measure through an abnormality range $[a, b]$, a security incident can be described using terms such as "high" or "low" which are more easily interpreted by humans [212]. By incorporating this concept to data mining, a value can contribute to the support of more than one fuzzy set. For instance, we can have a rule:

$$\{SYN = LOW,\ FIN = LOW\} \Rightarrow \{RES = LOW\} \qquad s = 0.5,\ c = 0.9 \qquad \ldots (10)$$

where SYN is the number of *synchronize* flags, FIN is the number of *finalize* flags, and RES is the number of *reset* flags in a certain time period. Instead of having numeric values describing these figures, fuzzy terms are introduced to give more flexibility to the rule. Fuzzy sets describing these terms (e.g., LOW, $HIGH$) would have to be created in order to map the input before performing intrusion detection [22].

15. **Support Vector Machines**. Support vector machines (SVM) are one of the most recent supervised-learning methods and is based on the addition of dimensions that allow for regression methods to be executed [135, 219, 130]. SVM's are constructs that perform binary classification. A set of input data points are mapped into multidimensional-space *vectors* using *kernel functions* so that they can be split into two classes.

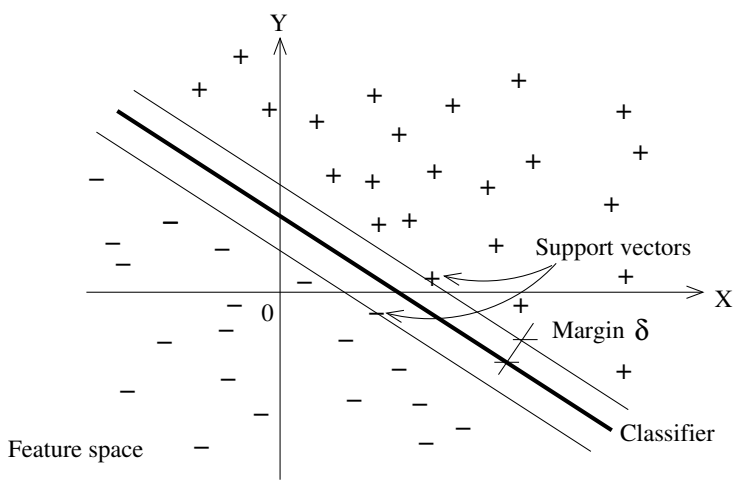


Figure 2.10: SVM splitting the feature space into two classes (they could represent normal and irregular behavior).

Some of these vectors are selected to define the boundary between the classes, and a hyperplane is computed through regression in order to split the input. It is then gradually adjusted by minimizing a *marginal-error distance* δ measured from the representative vectors (i.e. support vectors) to the hyperplane. It then becomes an optimization problem.

Just like neural networks, SVM's are trained in order to create classifier functions that help separate normal from abnormal activity. The training speed of SVM's is one of its most notable characteristics as well as its scalability (SVM's are "relatively insensitive" to the number of input data points [219]). They can learn large pattern sets and are able to scale better than neural networks.

Figure 2.10 shows an example of a SVM on a bi-dimensional feature space. Vectors $v_i = (x_i, y_j) \in X \times Y \times \{\pm\}1$ (which could represent packet information such as packet length and checksum) are classified by a linear function which has a variable marginal distance δ determined by support vectors close to the boundary. This margin should be maximized in order for classification to be accurate.

SVM's have been used to predict security events and also to classify system behavior and fingerprints [135, 219].

16. **Planning**. An approach presented in [66] involves the construction of partial-order plans (i.e., partial-order subsequences) that limit the exponential growth of the number of states in a state-machine representing an intrusion scenario [173].

    A planner builds "plans that achieve goals which describe situations that are desirable to an intruder" [66]. A monitor will search through the set of active plans and not through the set of all possible situations, hereby reducing overhead.

    A planning program for intrusion detection will use a knowledge base which contains dependencies between actions. These dependencies will be used to create alternative plans. For example, the arrangement of preconditions P_i necessary for achieving a certain state S can be given by an expression like:

$$(P_1, P_2, ..., P_k) \wedge (P_i \prec P_k) \wedge ... \wedge (P_j) \prec P_m \qquad \ldots (11)$$

    which could be used to represent a more concrete plan:

$$(cp, chmod, touch, mail) \wedge (cp \prec chmod) \wedge (chmod \prec mail) \wedge (cp \prec mail) \qquad \ldots (12)$$

    That means, a set of actions need to appear and they must keep some relative precedence among them. A plan is started with a start state and further steps are added as preconditions appear. A planner uses precondition sequences to generate attack schemes that can be then validated with Petri nets or a similar tool [66, 104].

    First order logic can be also used to capture a penetration plan using the same approach. For example:

$$\begin{aligned}
&\exists\ file_1, file_2, file_3, x \\
&\quad owner(file_1) = x \wedge \\
&\quad cp(file_1, file_2) \wedge \\
&\quad chmod(file_2, 4755) \wedge \\
&\quad touch(file_3) \wedge \\
&\quad mail(root, file_3) \wedge \\
&\quad cp(file_1, file_2) \prec chmod(file_2) \wedge \\
&\quad chmod(file_2) \prec mail(root, file_3) \wedge \\
&\quad touch(file_3) \prec mail(root, file_3) \wedge
\end{aligned}$$

$\Rightarrow attacker(x)$

which represents the conditions necessary to determine x is an attacker. Inference engines can be used to evaluate such plans with strong logic support. Each condition is evaluated and, in case all the conditions of a given plan are given, the system is able to tell intrusion is occurring.

17. **Bioinformatics sequence-alignment methods**. Sequence alignment is key to determine the similarity between two DNA sequences. Nucleotides or amino-acids are aligned in each sequence and a score is computed as to tell how well the sequences align with one another [39]. Pattern-matching algorithms originally developed for quaternary logic can be applied to determine the similarity between an input and an attack stream (Figure 2.11).

```
GTGACAAGCCAC--AAGAGG---CCTT--AG
--|---|--|-|---||---------|---|
--GGAGAC-CCCG-TAGGAA----GATAGCG

Key: | = match
     - = gap
```

Figure 2.11: Example of DNA sequence alignment [39].

These methods are adapted to inputs consisting of more than four event types (i.e., the number of system-call types on Linux environments are over 200). In general, alignment of observed commands with long attack patterns is rewarded without heavily penalizing gaps or misalignment. The reason is that a matched subsequence is highly relevant for intrusion detection compared to misaligned subsequences. Matrices and n-dimensional data structures are built in order to compute the matching score of an input against n attack patterns. As shown in Figure 2.11, protein patterns slide over the input similarly to other matching algorithms [104, 37].

The main difference with binary matching methods is that Bioinformatics algorithms involve more than two input symbols, which allow them to be extended more easily to the case of intrusion detection sequences where the input domain has more symbols.

The techniques above are some of the most common methods utilized to detect illicit activity in computer systems. Table 2.3 at the end of the chapter lists the reviewed techniques and classify them according to three main points: (1) their basic approach (anomaly-based intrusion detection (AID), misuse-based intrusion detection (MID)), (2) their foundation (probabilistic method or learning method), and (3) the data source they can use (audit information, network packets, or a different type).

2.1.4 Data collection and fusion

Data sources

Intrusion detection is based on information collection and processing. Multiple data sources are available to systems that try to identify suspicious symptoms. The first intrusion detection tools were *monolithic architectures* and would process data at a single location [49]. With the expansion of information systems, more data sources became available and intrusion detection systems diversified by specializing on specific data sources.

There exist two main aspects that can be taken into account for classifying intrusion detection systems [13]. By the *data* they use they can be: (1) host-based systems, or 2) network-based systems. By their *processing methods* they can be classified as: (1) centralized, or (2) distributed. We now describe these four categories which are relevant to the architecture proposed by this dissertation.

(1) Host-based intrusion detection

These systems are designed to protect a single server by scrutinizing large amounts of data contained in audit trails or **log files** as well as other system data structures. These files describe a system by keeping a record of every atomic service requested and every problem reported by the kernel and applications.

The design of audit trail modules and audit data reduction are research areas in information security that aim at improving the quality (i.e., expressiveness and regularity) of the information stored by historical records. Auditing was considered a secondary aspect of operating systems [4, 56] but now, due to the importance of computer security, there are numerous projects devoted to create consistent models that provide the operating system and intrusion detection tools with security-relevant information in a format that can be queried by automated entities [107, 90].

An audit tuple typically consists of fields such as user id, process id, log-on time, permissions mask, and even system-call numbers and system-call types. This information can be used to detect some types of attacks, but it does not suffice for catching others. Not all operating system events are recorded to audit files [56]. There are actions that actually take place but do not leave any trail of its execution. This constitutes a problem for auditing and intrusion detection systems since important data may be missing which would otherwise enable them to find an intruder.

The processing of audit data is expensive as the number of audit records generated by users and applications can be huge (in fact, for audit data to be suitable for intrusion detection it is typically necessary to enable *full-audit mode* on the host. This produces large files that are not easy to handle and prevents real-time execution forcing security checking to be performed off line. Additionally, audit data is prone to modification by an intruder as it is regularly stored on disk.

System data structures are another source of evidence available on the host. As a process executes, it modifies its environment through actions. The changes can be observed on user tables, open-file tables, disk reads and writes, the process table, etc. This sort of data is more reliable from the security perspective as it is more difficult to be tampered with (alterations to system data structures are more evident [125, 101]). The disadvantage of basing a design on this

data is that it is volatile information and hence it is necessary to cache it or put it in a buffer so that it can be analyzed and kept as evidence [46].

(2) Network-based intrusion detection

A network intrusion detection system processes network packets. In Ethernet networks, it sets the network interface in promiscuous mode in order to read all packets [170, 143]. Unlike host-based systems, the information a network-based system uses is not typically stored on a central repository and so it needs to be quickly filtered as it is being read (buffering methods are typically implemented in order to cope with high traffic loads [180]). In general, host-based intrusion detection can detect some attacks that network-based detection cannot and vice versa.

Hybrid systems are not strange [152, 77] but, in general, network-based intrusion detection have been more successfully implemented in working, commercial products [126, 2].

Although the network-based approach has some limitations with switched networks and encrypted connections (a system cannot look into an encrypted transfer unless it has the key), it allows for a prompt detection as the network packets received often contain the first indication of attack [143].

System-call traces, audit logs, and network traffic are the most popular data sources in intrusion detection. Other sources can be incorporated but their relevance is smaller. For instance, user command history can be used to tell what a user has been doing during the day [108, 109]. It stores command names and, possibly, the parameters passed to the shell interpreter. Session variables and alias expansion complicates this approach as well as the fact that users can remove or disable their own command history preferences. Keystroke recorders can also be used [171] but the information that can be captured with them is limited as many attackers use scripts to automate their tasks and this prevents keystroke monitors from succeeding.

Table 2.1 shows some of the most popular intrusion detection systems classified into host-based and network-based systems [223, 13]. Notice that many projects have involved host-based models, however, network-based intrusion detection has been more effective and more widely accepted [2, 126].

Data fusion for intrusion detection

Historically, intrusion detection systems have been integrating more and more data sources to collect information. Data collection can be performed locally by accessing information stored on the host. Research shows, however, that by correlating information from a number of sources, detection accuracy increases [1, 185].

DEFINITION 12

Data fusion. Study of the integration of information records extracted from multiple, heterogeneous sources [16, 185].

Data warehousing tools as well as Web search engines are two applications that have proved the benefits of integrating data. Intrusion detection also deals with this problem and there have been efforts to design architectures that collect information from many distributed locations

Network-based	Host-based
ACME (Advanced Counter Measures Environment) [28] AAFID (*Autonomous Agents for Intrusion Detection*)[224] AFJ(*Anzen Flight Jacket*) [8] BlackICE Sentry [20] Bro [152] CaptIO [29] Centrax [141] CERN NSM (*Network Security Monitor*) [133] CyberCop [138] CyberTrace [44] DefenseWorx [139] DIDS (*Distributed Intrusion Detection System*)[187] Dragon [58] EMERALD (*Event Monitoring Enabling Responses to Anomalous Live Disturbances*) [140] eTrust ID [26] GrIDS (*Graph-based Intrusion Detection System*) [194] Hummer [158] LANguard SELM [70] LISYS (*Lightweight Intrusion detection SYStem*) [82] Manhunt [198] NetProwler [199] NetRanger [36] NetSTAT (*Network-based State Transition Analysis Tool*) [209] Network Security Agent [88] NFR (*Network Flight Recorder*) [164] NID (*Network Intrusion Detector*) [52] NSM (*Network Security Monitor*) [77] OpenSnort Sensor [191] RealSecure [89] SecureNetPro [129] Shadow [145] Snort [170] T-sight [57]	AAFID (*Autonomous Agents for Intrusion Detection*) [224] ADS (*Attack Detection System*) [167] AID (*Adaptive Intrusion Detection*) [188] ALVA (*Audit Log Viewer and Analyzer*) [131] AppShield [176] ASAX (*Advanced Security Audit Trail Analysis*) [75] CARDS [218] Centrax [141] CIDS (*Cooperative Intrusion Detection*) [215] CompWatch [53] CyberCop Monitor [138] CylantSecure [216] DIDS (*Distributed Intrusion Detection System*) [187] DPEM (*Distributed Program Execution Monitor*) [102] Dragon [58] EMERALD (*Event Monitoring Enabling Responses to Anomalous Live Disturbances*) [140] Entercept [123] eTrust Audit [26] FormatGuard [40] GrIDS (*Graph-based Intrusion Detection System*) [194] Haystack [175] HP IDS/9000 [79] Hummer [158] Hyperview [46] IDA (*Intrusion Detection and Avoidance*) [189] IDES (*Intrusion Detection Expert System*) [50] IDIOT (*Intrusion Detection In Our Time*) [105] Intruder Alert [197] Janus [72] JiNao [95] Kane Security Monitor [55] LIDS (*Linux Intrusion Detection System*) [85] MIDAS (*Monitoring, Intrusion Detection, Administration System*) [182] NADIR (*Network Anomaly Detection and Intrusion Reporter*) [81] NIDES (*Next-Generation Intrusion Detection Expert System*) [5] NSTAT (*Network State Transition Analysis Tool*) [97] pH [190] POLYCENTER [47] PRCis [160] Psionic HostSentry [35] Psionic PortSentry [35] RealSecure [89] SecureNetPro [129] Shadow [145] StormWatch [148] Tripwire [100] UNICORN [34] USTAT (*Unix State Transition Analysis Tool*) [87]

Table 2.1: Classification of intrusion detection systems by input type [223, 13].

and consolidate it into a main processing unit [185, 224, 187]. These systems are known as **centralized processing systems**.

Security-relevant evidence can be captured from sources such as network packets, audit log files, system tables, integrity checkers, vulnerability scanners, anti-virus software, and security monitors. There is already sufficient data. The challenge is in identifying the mechanisms to efficiently process this data so that attack signs can be found [18].

Some research teams have built tools that integrate host information and network data [187, 194]. A great deal of effort has been put into developing structured ways of correlating these two types of information so that attacks can be more accurately detected. It is important to mention that even when it is theoretically a good idea to use as many data sources as possible, the diversity of data formats and services significantly complicates this task [185].

Unlike centralized processing systems, there are others that collect evidence in a distributed fashion but do not centralize the processing [187, 194, 95]. These systems are known as **distributed processing systems**. Every component processes its own data and, by means of message passing, they share their findings in order to reach conclusions regarding the security state of the system. The complexity of these systems ends up impacting performance and the overhead they impose is typically unacceptable [92]. The distributed approach is more resilient to failure compared to the centralized model.

Data fusion can be performed by abstract layers. These compose a communication structure in which data-capture modules mechanically send the information they observe to data correlators [81, 5, 140]. These correlators have more intelligence built in and are responsible for the actual data processing. Many intrusion detection systems support hierarchical structures like this and consequently suffer from central-point-of-failure problems [224, 15, 78, 215].

These systems constitute the area of *distributed intrusion detection* which analyzes challenges such as *spam* and distributed denial of service [111, 187]. In recent years, viruses, worms, and distributed attacks have encouraged the design of new solution alternatives. Data fusion has represented a natural way of dealing with the problem, although some data representation and sharing issues need to be resolved in order to make distributed intrusion detection a more reliable and robust solution [112, 187].

Table 2.2 classifies intrusion detection systems into *centralized processing* and *distributed processing* systems according to the data analysis structure they have [223].

2.1.5 Challenges of current models

Intrusion detection technologies still have significant improvement areas that need to be worked out in order to create effective and efficient products. The difficulty of this security problem resides in the difficulty to accurately identify anomalous behavior given the amount, correctness, location, availability, and structure of the data that needs to be analyzed. Some of the main challenges relevant to the topics of this dissertation are described next.

1. *False alarms.* Many intrusion detection tools suffer from detection inaccuracy [2, 90, 4, 126, 13]. The high rate of false positives and false negatives that intrusion detection tools generate does not help system administrators simplify their daily tasks. On the contrary, users may eventually opt for disabling this software rather than having to interpret the stream of alarms reported by the security system that may, in fact, not be reflecting the truth. This is a major limitation of intrusion detection technologies and is the priority for multiple data fusion and model-based projects.

Centralized processing	Distributed processing
ACME (Advanced Counter Measures Environment) [28] ADS (*Attack Detection System*) [167] AID (*Adaptive Intrusion Detection*) [188] ALVA (*Audit Log Viewer and Analyzer*) [131] AppShield [176] ASAX (*Advanced Security Audit Trail Analysis*) [75] BlackICE Sentry [20] Bro [152] CaptIO [29] CERN NSM [133] CIDS (*Cooperative Intrusion Detection System*) [215] CompWatch [53] CyberCop Monitor [138] CyberTrace [44] CylantSecure [216] DefenseWorx [139] Entercept [123] eTrust Audit [26] Haystack [175] Hyperview [46] IDA (*Intrusion Detection and Avoidance*) [189] IDES (*Intrusion Detection Expert System*) [50] IDIOT (*Intrusion Detection in Our Time*) [105] Intruder Alert [197] Janus [72] Kane Security Monitor [55] LANguard SELM [70] LIDS [85] Manhunt [198] MIDAS (*Monitoring, Intrusion Detection, Administration System*) [182] NADIR [81] NetProwler [199] NetRanger [36] Network Security Agent [88] NID [52] NIDES (*Next-Generation Intrusion Detection Expert System*) [5] NSM (*Network Security Monitor*) [77] NSTAT (*Network State Transition Analysis Tool*) [97] OpenSnort Sensor [191] pH [190] POLYCENTER [47] PRCis [160] Psionic HostSentry [35] Snort [170] T-sight [57] Tripwire [100] UNICORN [34] USTAT (*Unix State Transition Analysis Tool*) [87] Wisdom & Sense [205]	AAFID (*Autonomous Agents for Intrusion Detection*) [224] AFJ (*Anzen Flight Jacket*) [8] CARDS [218] Centrax [141] DIDS (*Distributed Intrusion Detection System*) [187] DPEM (*Distributed Program Execution Monitor*) [102] Dragon [58] EMERALD (*Event Monitoring Enabling Responses to Anomalous Live Disturbances*) [140] FormatGuard [40] GrIDS (*Graph-based Intrusion Detection System*) [194] HP IDS/9000 [79] Hummer [158] JiNao [95] LISYS (*Lightweight Intrusion detection SYStem*) [82] NetSTAT (*Network-based State Analysis Transition Tool*) [209] NFR (*Network Flight Recorder*) [164] RealSecure [89] StormWatch [148]

Table 2.2: Classification of intrusion detection systems by their data analysis structure [223, 13].

2. *Amount of data.* The amount of data available within a single host can make the intrusion detection problem intractable. In the case of network-based intrusion detection, the traffic

going through a gateway, for instance, can be prohibitively large making it practically impossible for a security scanner to check every single packet without dropping some of them [143, 4]. Packet dropping severely damages the effectiveness of this type of protection tools.

Similarly, audit facilities can generate very large log files which limits the likelihood of performing real-time processing [4]. What this does is that audit analysis has to be moved to the end of the day for off-line checking, which is not as desirable as real-time detection.

Intrusion detection is a needle-in-the-hay type of problem: the amount of "normal" activity data is huge compared to the pieces of irregular activity that need to be found.

3. *Performance.* Data filtering and screening is a computationally demanding task that is central to intrusion detection [126]. Providing a system with real-time monitoring capabilities implies good search and pattern matching algorithms as well as efficient buffering of network traffic, audit trails, etc. which guarantees no data piece goes through without being inspected. Given the detailed checking needed by intrusion detection, a lot of resources are allocated and consumed and the performance of the hosting system ends up being impacted. As a consequence, heavy intrusion detection software and business processes cannot cohabit and the former is sometimes disabled to give full privileges to priority services [60].

4. *Switched networks.* Modern networks utilize more intelligent devices to route packets from one location into the other. Many intrusion detection prototypes were based on the ability to configure a network device in promiscuous mode (e.g., [152, 170, 5]) in a way that the monitoring mechanism could easily see and analyze all Ethernet packets. This allowed the system to correlate packets addressed to different hosts connected to the same segment, and also to protect several hosts at the same time. With modern network technology, this approach may no longer be as successful.

   Switches only forward packets to its intended destination to optimize bandwidth use [38] (i.e., broadcasting is limited). Now, the number of security software instances that need to be installed is larger and network-based systems need to be strategically placed so that they are able to protect as many hosts as possible inside a LAN. The filtering functions of a switch which represent a performance advantage from the bandwidth perspective, are in fact a challenge to the protection of networked systems [126, 2].

5. *Encryption.* Encrypted data cannot be easily interpreted unless a decryption key is available [179]. Information technology has witnessed the development of cryptographic protocols and tools that protect the privacy of data. File transfers, interactive sessions, and email can all be protected with encryption. Although this secures the information they handle, it renders further security analysis impossible – necessary in the case they transport illicit material [142]. As more companies become security-conscious the use of encryption will increase [169, 124, 48]. Intrusion detection tools are currently inadequate to deal with this problem as they are blind to encrypted communications.

6. *Security of intrusion detection technology.* Security tools are expected to be secure themselves – otherwise they could not be trusted and their results would be worthless [2, 42]. As software products, intrusion detection systems face the same security challenges other applications do (e.g., poor implementation and weak design). In fact, not much attention has been put

into the secure design of this type of software [134]. Only a few systems can be considered moderately secure but the rest is prone to direct or evasion attacks [13, 161]. This is an aspect that cannot be ignored when designing protection systems.

7. *Reaction to incidents.* Although it is not part of the traditional definition of intrusion detection [4, 60, 126], response to incidents is a very desirable feature that a few intrusion detection systems have actually tried to automate [72, 215]. Taking action based on incorrect information can have serious consequences especially when done on production system. The problem of automated response has remained unexplored due, in great part, to the low detection-accuracy levels intrusion detection tools achieve nowadays [2, 13]. This area will develop when detection algorithms become more precise and when intrusion alarms reveal meaningful security events.

2.2 Recent trends in information security

Over the last five years, protection tools, security threats, and cyber-crime have presented behavioral patterns that reveal changes in the way computer and information security is conceived. These trends need to be studied and fully understood so that new technology can be developed to effectively mitigate the damages caused by worms, viruses, hacking, espionage, and other modern threats.

1. *Decreasing average technical expertise in attackers.* The number of security incidents has been growing steadily with the years reflecting the expansion of the Internet. Recent cyber-attacks are very serious and extremely sophisticated in nature [124]. This is in part due to the increasing complexity of software along with the fact that more people are making use of security software which makes break-ins more difficult [169, 124].

   The average technical knowledge of attackers, however, has decreased (Figure 2.12). It is no longer necessary to be a systems expert for launching an attack on the Internet. A few people create scripts and automated tools which are eventually obtained by a larger number of novice users. They may not know the details of how the exploits work, but have the curiosity of seeing what they actually do. These users represent the most dangerous group as they do not know the consequences of the attack tools they execute [42].

2. *Accelerated growth in the number of vulnerabilities and attacks.* One of the clearest trends in information security is the fact that more and more vulnerabilities are discovered everyday and, consequently, more attacks that exploit them also appear in the scene [169, 124, 48]. Figure 2.13 contains some numbers that show this trend.

   The purpose of the release of vulnerability information is to make users and administrators aware of the existing threats. However, this information is also used by attackers that devise new techniques to abuse systems.

3. *Increasing number of external attacks.* Years ago the rule was to see internal attackers abusing their privileges to compromise a local host. External attacks were not as common, but that has changed [201]. With the connectivity provided by modern networks more people

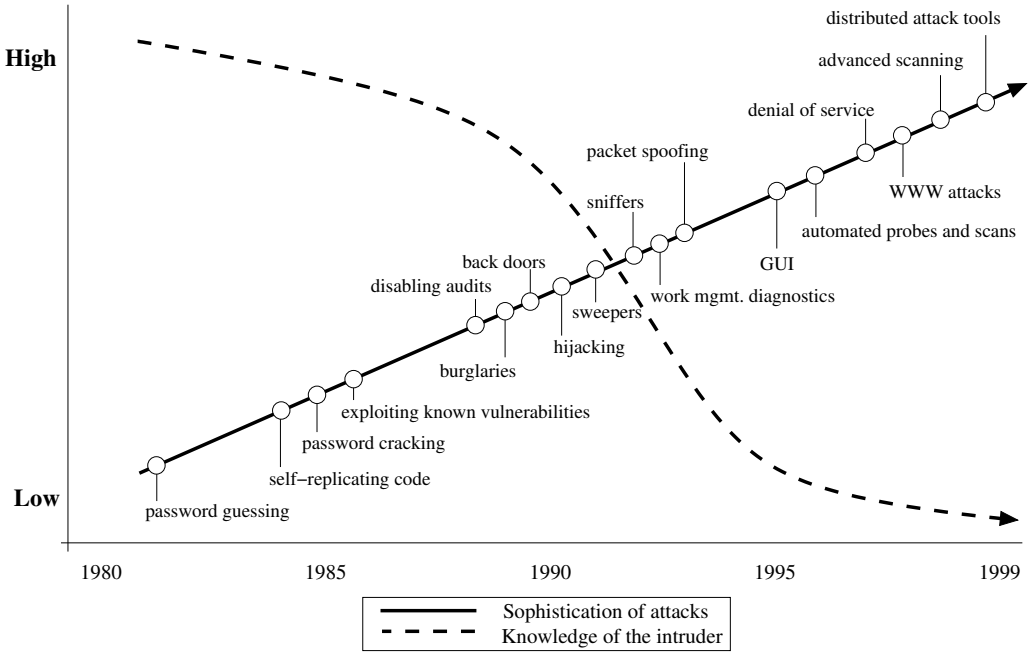


Figure 2.12: The increasing sophistication of attacks and the decreasing expertise of users [2, 126].

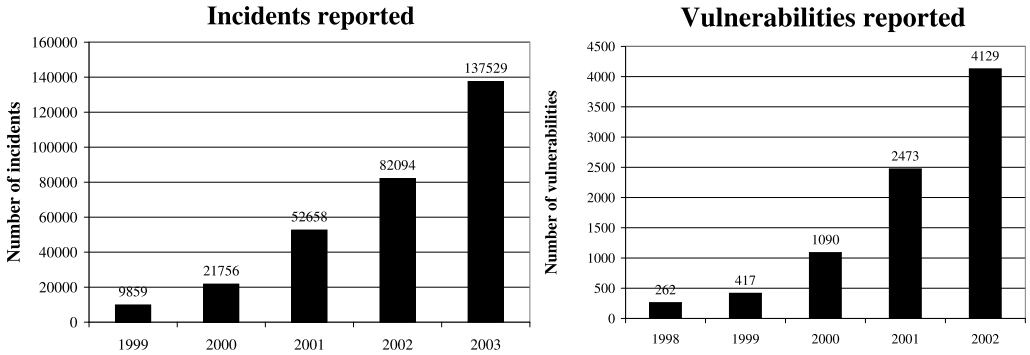


Figure 2.13: Number of security incident and number of vulnerabilities reported per year to the CERT–CC (*Coordination Center, Computer Emergency Response Team* [32]).

have access to the Internet and have the potential of becoming an attacker. As a result, the number of attacks coming from the outside, according to the information reported by many companies, has significantly increased [169, 165, 2, 42] in parallel to the discovery of remotely-exploitable vulnerabilities which the on the raise [201]. Given the visibility administrators have of their own systems, internal users are easier to catch, that is why this trend reveals an increasing number of incidents originated by outsiders who can very well hide through

connection hopping in the net [118, 117].

4. *Increasing use of security technology.* Firewall software, authentication systems, and anti-virus technology are three common weapons security administrators usually have on their belts [169, 165]. According to recent surveys, intrusion detection programs are being added by more people as a component of the typical corporate security suite (Figure 2.14). Although this demonstrates the importance of security monitoring and encourages the creation of better tools to help identify illicit activity, vulnerabilities and exploits are on the raise using more complex methods to defeat the security mechanisms in place.

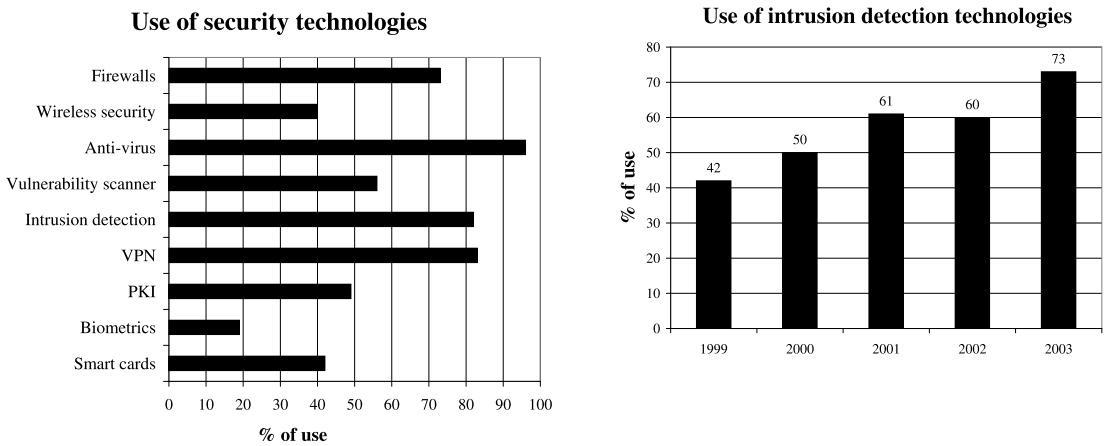


Figure 2.14: Use of information security technologies (*2003 Global Security Survey, Deloitte & Touche* [48]) and intrusion detection technologies (*CSI/FBI Computer Crime & Security Survey 2003* [169]). The charts show the percentage of surveyed companies that report using this type of software.

5. *Increasing availability of automated attack tools.* Recent statistics on software vulnerabilities and security reports coincide by indicate that an increasing majority of attacks on the Internet are being executed with the use of automated attack tools [169, 165]. Years ago there were not many websites that stored programs that exploited vulnerabilities automatically. Given the openness to report software bugs as a security strategy [163, 10] more users have access to the details of how to build programs that automate systems probing, scanning, and the actual exploitation of security wholes [103]. The increasing number of attack tools available on the Internet coupled with a low average of technical expertise from users is a combination that represents a severe challenge to information security researchers and practitioners.

6. *More Distributed Denial of Service Attacks.* The number of denial of service attacks (i.e., attacks aimed to exhaust the resources of a host by flooding it with service requests) orchestrated from remote locations keeps increasing (Figure 2.15 ;[169]). These attacks sometimes compromise a group of machines that will be later used to launch an attack using remotely-managed tools toward a common target.

The inability of current network protocols – in particular the TCP/IP family – to preserve the integrity of packets makes of this attack one of the most difficult to defend against [111, 161]. Every field in a packet can be altered in such a way as to evade security mechanisms, which complicates the task of packet filtering and source back-tracing. The latter has been studied for several years now and it is still considered a major challenge due to the underlying network infrastructure we use to keep the Internet running [222].

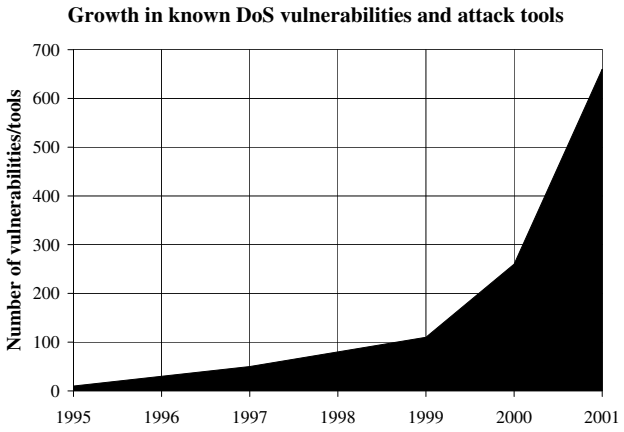


Figure 2.15: Number of known Denial of Service (DoS) vulnerabilities and attacks per year (*PestPatrol's detection database, PestPatrol Inc. 2002* **[155]).**

7. *Network hopping for attack relaying.* Attackers protect their identity by hopping among a series of hosts before launching an attack. This connection chain hides them from administrators and law enforcement as it is very difficult to identify the exact location from which a connection is truly coming [118, 11]. Additionally, proxies and anonymity tools can be used to complicate the task of identifying the root of an incident, an important aspect in a cyber-crime investigations [168, 73].

8. *Increasing exploitation of recent vulnerabilities.* The time between the discovery of vulnerabilities and the appearance of a number of attacks exploiting it has been decreasing [124]. In the past, two or three-year-old vulnerabilities were frequently used to compromise systems which had not been patched. The vulnerability-age range was considerably large. Now, more than 50% of the attacks involve vulnerabilities reported during the last five months. Two factors have influenced this trend. Firstly, vulnerability information is more readily available on the Internet and an attacker planning to compromise a system can look for the most recent vulnerabilities to increase the odds of spotting vulnerable systems somewhere. Secondly, more companies are becoming security-aware and this has reduced the number of systems that have old bugs or misconfiguration problems [169].

2.3 Connection back-tracing

Given a set of hosts $H = \{h_1, h_2, ..., h_n\}$, with $n > 2$, a user could create a connection chain of m hops as follows: he could connect from host h_i into h_{i+1}; from h_{i+1} into h_{i+2}; from h_{i+2} into h_{i+3}; and so on, until finally connecting to target host h_{i+m}. The problem of finding the hosts that compose the connection chain and identifying the root host h_i given H, h_{i+m}, and user name u on h_{i+m}, is known as the **connection back-tracing problem**.

For a small $n = |H|$ this problem is straightforward. Just a few hosts need to be analyzed in order to identify the source of a connection sequence. However, as n becomes larger (e.g. $n \gg 2$), the complexity of the problem increases since a larger number of hosts in H multiplies the possibilities of creating a more confusing connection path.

2.3.1 Limited troubleshooting visibility

System administrators have limited access to information regarding who is connecting to one of their servers. On a LAN, administrators typically have login access to all servers, which makes the tracing process easier. However, for connections coming from the outside of the LAN, they are not able to freely access all other servers for scrutiny. The visibility of this type of connections reduces to one link in the connection chain. That is, an administrator can only see the terminal number or the IP address/host-name of the immediate host from where users are logging on. He is unable to tell, however, whether that host is in fact h_i (the workstation of the attacker). In order to find out whether it is the root host or not, he will have to contact the administrator of the remote system who, in turn, will have to perform a similar checking.

This job is cumbersome and requires time, communication, and trust from the parties involved. For the case of a connection that has been closed already, audit information will have to be examined in order to find the nodes of the chain that lead to the origin of the connection. This off-line sort of tracing has an important flaw: it counts on the existence and integrity of audit log files. For clean systems this would not be a problem but for systems that have been compromised, and in the case of a connection chain created to perpetrate an attack this could probably be the case, these files could have been tampered with during a previous break-in [222]. On the other hand, if a connection is still alive, i.e. there is a packet flow between source and destination, information contained on system tables and network packets can be used to trace the user while he is on-line.

2.3.2 Related projects

There are some projects that touch on the problem of connection-route tracing. One is the method proposed in [222]. This model implements network traffic monitors that perform session tracing once a server has been compromised. A system that computes deviations from the traffic observed at two different hosts helps determine if these hosts were used in the same connection chain. Since checking network traffic at a node involves huge amounts of data, multiple packet monitors permanently filter and record specific system activity to build their own logs. These files are analyzed by a data correlator to identify the hosts of a connection chain. It is important to notice that this system deals with the problem of deleted logs by creating its own records. These records, however, may be the target of the same type of attack.

IDA (Intrusion Detection Agent system) [11] is another system that detects the origin of an information exchange related to an incident. It is primarily an intrusion detection system that employs mobile agents to detect local attacks. From a main host, investigation agents are sent over to the requester that needs an integrity check. These agents use the MLSI (*Marks Left by a Suspected Intruder*) strategy to reduce the amount of data needed to flag system activity as anomalous. In order to collect further information on a break-in, IDA does passive route tracing using audit log data. The structure of the tracing method is not specified by the authors but its agents broadcast connection evidence to their peers as a way of identifying possible hosts involved in the attack.

The *TBAIR* system (*Tracing-based Active Intrusion Response*) [213] is a network-based intrusion detection tool that, unlike most traditionally-passive intrusion detection systems, adds active response to security events using a technique called *Based on Sleepy Watermark Tracing*. TBAIR tries to attack the root of the vulnerability exploiting problem by locating the originators of the incident to hold them accountable for their intrusions and their method is based on the analysis of evidence collected from the network.

Thumbprinting [195] is a technique that places several processes across the network in order to capture activity signatures or "thumbprints". This method does not require having processes on all hosts and, like in the case of [222], it replicates packet information for future review. This system is based on the fact that the packet content of a session passing though a set of hosts is invariant and can be compared in order to identify a host in the chain. This system is capable of detecting the root of a connection but it is not designed to identify the exact hosts that are part of the chain.

2.4 Agent-based intrusion detection

Multiple artificial intelligence techniques have been applied to information security problems. There are authentication systems that use artificial neural networks to perform fingerprint and retinal-pattern recognition in order to identify the users of a system with a high precision level [18, 137, 156]. Fuzzy logic, decision trees, and support vector machines have also made possible the creation of face recognition and keystroke profiling tools to enhance access control, a critical security component of a system [171, 120, 219]. Redundancy through replication using autonomous agents has been also studied in order to strengthen the robustness of high-availability platforms [63, 122] (Figure 2.16).

Agent-based software in many different areas have helped understand the security requirements of services and applications [14, 21]. The field of intrusion detection has not been the exception and artificial intelligence has also contributed with some solutions [224].

Data mining software, for instance, has been built to monitor the security of a system and has been capable of successfully identifying electronic fraud and, in some cases, back-trace intruders [114, 113, 11]. A number of anomaly detection techniques have been implemented using neural networks and support vector machines [135, 174, 27, 46]. Their performance is still a challenge but further research in this young area is likely to produce better solutions.

Inside a host, different data sources need to be consolidated in order to discover distributed attacks. Outside of the host, networking has allowed the integration of diverse platforms with the purpose of supporting electronic services and resource sharing. This integration has expanded

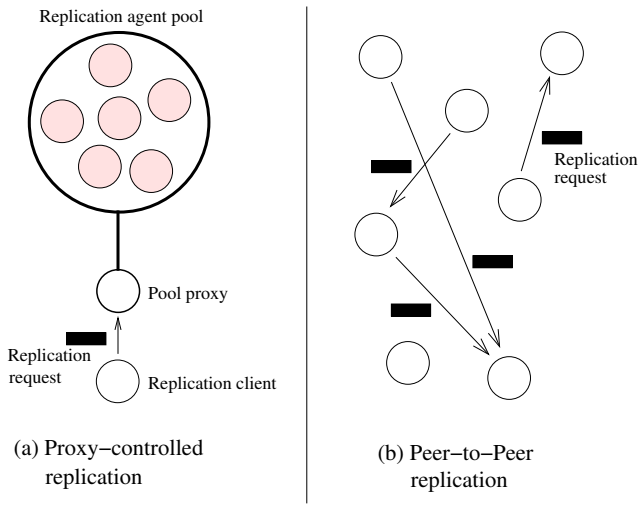


Figure 2.16: (a) Replication is performed by a pool of agents which are contacted through a proxy agent. (b) Replication can also be peer-to-peer.

the coverage area for which a monitoring module is responsible. It is no longer a single server or application what needs to be protected. Every single component can be vulnerable to attack and needs to be secured. Even if one of these components has not been apparently compromised, the activity observed by its neighbors may indicate the opposite – a victim does not always have enough visibility as to infer it is being attacked [60].

Intrusion detection systems need to be able to correlate input from multiple local or distributed sources in order to prevent attackers from gaining unauthorized access to it. The data integration nature of intrusion detection has made researchers look at the agent and multiagent approach in the search for more and better solutions [224].

2.4.1 Agents

The definition of agent we use throughout this dissertation is the following:

DEFINITION 13

Agent. A computational entity with a certain degree of autonomy that perceives its environment through sensors and modifies it through effectors [173].

Some authors suggest agents need to be independent enough as not to require intervention from humans (otherwise they could be considered mere software applications [214, 3]). In order to reach that independence or autonomy, it is necessary for agents to be able to decide what action to take next based on their input perceptions. This decision may be influenced by a utility or goal function that gives meaning to the agent (in the case of security monitoring agents, rarely are these functions explicitly defined [41, 224]). If all of the agent's actions are selected as to reach this objective based on the available information, the agent is said to be rational [172].

Rationality is another desirable characteristic of agents and recent publications emphasize the issue of how to determine whether or not an agent is trying to achieve its goal, and how well it is doing in the process [173]. Performance functions are employed to evaluate the agent's performance and a rational agent will always try to maximize this function regardless of the uncertainty of the environment and just based on its perceptions and prior knowledge.

Sensorial information (i.e., perceptions) is key to an agent's success. It allows it to make decisions and acquire knowledge. Learning is a process through which an agent will use perceptions from the past to make well informed decisions at present predicting the future [159, 173]. The knowledge-base of an agent captures information meaningful to the utility function and that helps an agent achieve its objectives.

There are other characteristics that, depending on the application, are desirable in an agent. Such is the case of mobility, persistence, initiative, communication, coordination, and sociability [173]. The latter three represent important considerations of what researchers have called *distributed artificial intelligence* [214].

2.4.2 Multiagent systems

An agent can be designed to work alone but in many practical scenarios multiple agents are executed to accomplish a single task. Sociability is the ability an agent has to communicate and interact with its peers in order to accomplish such task and **Distributed Artificial Intelligence** (DAI) is a discipline that studies the use of collections of agents to perform joint tasks [172, 151].

DAI comprises two broad issues: (1) finding ways to split a certain problem into subproblems so that it can be solved more quickly by having several entities working concurrently, and (2) utilizing multiple existing agents to solve a problem. The slim line that divides these domains has in fact created two different research areas, namely, *distributed problem solving* and *multiagent systems* [146].

Distributed Problem Solving (DPS) has to do with the analysis of the structure of problems and their potential solutions. Some problems can be divided into into several smaller problems whose solutions can be obtained separately and then combined to resolve the main problem [116]. The problem could also be transformed into an equivalent representation that allows parallel processing. Distributed problem solving has an algorithmic perspective that differs from the Multiagent Systems approach to artificial intelligence [54, 146].

Multiagent Systems (MAS) are primarily concerned with the behavior of a collection of agents. It focuses on developing mechanisms to get groups of agents to cooperatively accomplish a goal they could not solve individually. This includes problems like collaboration, negotiation, coordination, and knowledge sharing where agents need to use their skill-set to communicate and solve, as a team, a common problem [14, 146, 25, 86].

From a different perspective, we can also define DAI as a subarea of MAS. Agents in DAI are mostly cooperative and friendly. MAS studies this basic behavior plus more complex cases in which agents negotiate, complete, and can even become selfish..

Information security researchers have been interested in agents for two main reasons. Firstly, agents are implemented in software and, as such, are prone to attack. This constitutes an interesting research problem as the mobility and communication features of agents complicate

the task of securing information and guaranteeing agent availability [74, 122, 157, 45, 62]. The problem of agents trying to compromise their peers has been studied. However, the fact that the actual execution environment can be hostile represents a threat to the integrity of agent-based software and is a topic that has proved challenging and more interesting [177, 178, 19]).

Secondly, and being the aspect that occupies this dissertation, a multiagent system can serve the purpose of carrying out security-related tasks [78, 93]. System administrators take advantage of their experience in order to identify actions that could uncover an intrusion. That experience coupled with their ability to correlate information and draw conclusions, is something that needs to be built in computer intrusion detection systems. The automation of this process through monolithic intrusion detection programs [49, 182, 50, 75] has due to challenges like the ones we presented in previous sections (see Section 2.1.5). For that reason, there have been different architectures that integrate agents to accomplish the goal of providing a host with monitoring capabilities. Following a description of the components of some of these architectures.

2.4.3 Multiagent components of intrusion detection architectures

Some agent-based architectures have been implemented in the past for the problem of intrusion detection [25, 224, 158, 15]. They all share structural aspects and component types that are mentioned next and that constitute the base of the system presented here.

Agent-based intrusion detection architectures

The term "agent" has been overused in many information technology areas [21] including computer security. The aim and characteristics of agents have not been yet fully understood by, and integrated with neighbor areas which are often attracted by what this promising computing approach has to offer.

Intelligent capabilities such as learning, autonomy, and pro-activity are rarely seen in agent-based intrusion detection implementations. The architecture by Balasubramaniyan et al. [15], for instance, displays communication between processing entities limited to the structure of a message-passing mechanism. It does not enable agents to do inference with the information contained in the messages they receive, and the module interdependence of the system prevents even the most basic levels of agent autonomy.

Crosbie and Spafford [41] also designed an architecture to implement system monitoring. Some of their agents perform data collection and analysis but they were not designed to use seen events or to evaluate the success/failure of their re/actions (i.e., they do not accumulate any sort of experience) to improve on their own performance. The Hummer project [158] and the AAFID architecture [224] are agent-based intrusion detection environments where data is captured, inspected, and correlated to perform intrusion detection (Figure 2.17). Some of the hierarchical structures used to integrate the components of these models, however, severely limit agent autonomy and constitute potential attack points and/or bottlenecks that cannot be present in robust security software.

An element required for some agent functions is a layer which provides a common interpretation of the environment. This layer is missing in all of the agent-based intrusion detection projects reviewed in the literature. An ontology could provide security agents with a better

Sophos Ex. 1007
Page 59 of 216

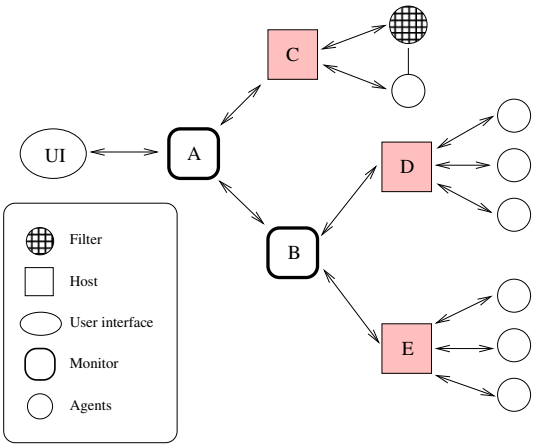


Figure 2.17: Hierarchical component of a multiagent intrusion detection architecture [224].

foundation for more naturally integrating components and sharing information while enabling the development of more adaptable and useful intelligent software [23].

DEFINITION 14

Ontology. The specification of the objects, concepts, and relationships in an area of interest [214].

By having an ontological model as the underlying element of a multiagent platform,

Agent communication and development would benefit from an underlying ontological model that gives the agent platform solid ground for building more ellaborate functionality for intrusion detection tasks. Figure 2.18 depicts a logical composition of blocks representing fundamental components of an intrusion detection system based on intelligent software. When built on top on an ontology, agents can be more easily maintained as the overall functioning of the system is guided by the definitions and relationships stored on the ontology.

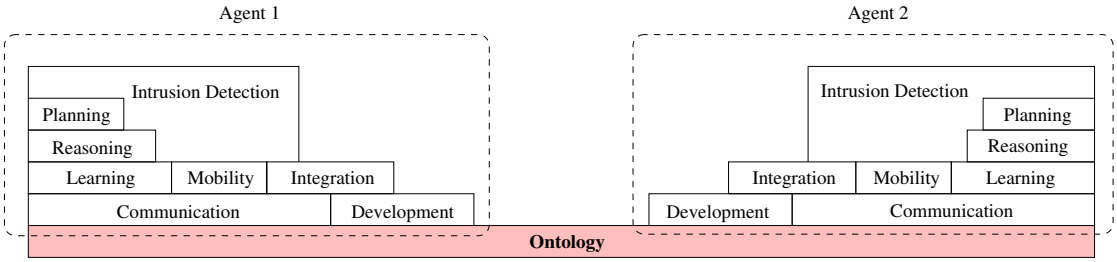


Figure 2.18: Enabling high-level agent functions through an ontology layer.

Agent-based intrusion detection architectures

There is a series of tasks that are necessary to perform intrusion detection and which can be distributed among a group of agents. They can be summarized in three major tasks: *data collection*, *integration*, and *correlation* or *reasoning* [4, 192, 103].

- **Sensors**. Data collectors are usually referred to as *sensors* in intrusion detection [215, 78] (notice that sensors are also pieces of code used to detect anomalies when embedded into a program; see [223] for details). Sensors can be placed at different locations within a network in order to capture information that may contain signs of intrusion. They could be agents that simply read and forward to a central authority all the information they receive, or they can be enabled with filtering capabilities that allow them to pre-process the data before sending it to other agents. Further more, sensors can have intelligent functions built in – through learning and reasoning techniques, e.g. – so that they not only reduce and filter, but also classify and check for consistency the data they make available to its peers [41].

- **Integrators**. Integration agents receive information from others, condense it, and re-write it using formats higher-level agents can understand. The role of integrators is to relieve other layers from structuring and sanitizing the data, as well as to take over the role of querying every sensor that may have acquired interesting information security-wise. Interrupt-based or polling methods are equally used for the communication of these systems [192, 224].

- **Correlators**. The agents that implement the core of intrusion detection algorithms are the *correlators*. Their objective is to use the data they are sent in order to detect signs of intrusion. In order to do this, they need to be equipped with mechanisms that allow them to create relationships between the pieces of information they have. Decision trees, inference models, and neural networks are just some of the techniques that have been utilized by this type of agents to accomplish their goal [174, 27, 215, 45].

There are several ways to organize these agent classes. Each agent can be responsible for reaching a specific subgoal whose results are to be sent to other agents in charge of processing low-level information. Most existing intrusion detection models use a hierarchical structure like this [224, 15, 106, 78, 41] (Figure 2.17). However, recent architectures have proposed peer-to-peer communication where a self-contained agent is capable of performing several task by itself [92]. A hierarchical structure is more efficient from the performance point of view but, since the number of agents that compose a security architecture is typically small [15, 224, 41], a peer-to-peer system provides desirable advantages from the availability point of view that make it a good alternative.

2.5 Taxonomies and ontologies in information security

Systems security has struggled with the fast-paced expansion of the Internet. As a result, the formalization of many concepts is still missing as it is the case with intrusion detection which lacks structured representation, communication, and classification models. This section describes some relevant efforts in these areas related to the objectives and motivation of the present dissertation.

- **Anderson's attacker classification**. One of the earliest classifications to appear in the area of intrusion detection was the definition of attacker types [6]. It divided attackers into *external penetrators* (i.e., outsiders), *masqueraders* (i.e., users who steal identities from others), *misfeasors* (i.e., those who abuse the privileges they have been granted), and *clandestine users* (i.e., those who are given access by another user but are not known by system administrators and audit records). This classification is important since it was one of the initial efforts to understand the phenomenon of illicit system use.

- **Kumar's taxonomy of computer intrusions**. This work states that intrusions are similar among one another and can be grouped into generic classes described by patterns [103].

  This project presents a hierarchy for classifying intrusion signatures based on the structural interrelationships among the events that compose them. For instance, a file-write event could be preceded by a permission-change event at the beginning of a particular intrusion scenario and the sequence of these two events – with their corresponding parameters and the existence of some execution variables – could be common to more than one intrusion type. A signature can be created as to represent this precedence and all intrusions that matched the signature will be put together in the classification tree.

  The events defined by this taxonomy are composed of lower-level elements such as system calls and system constructs (e.g., i-nodes, permission masks). This taxonomy shows how colored Petri nets can be used to model the classification and matching system (similar signatures share the same Petri net; [105]). Even though this taxonomy was not designed to classify vulnerabilities but the signatures to detect when they are exploited, it became the foundation of a vulnerability database [128].

- **CVE classification of vulnerability types**. The *Common Vulnerabilities and Exposures* initiative (CVE) is a list of standardized names for vulnerabilities and other information security exposures [121, 33]. The goal of CVE is to make it easier to share data across separate vulnerability databases and security tools [127]. The content of CVE is a result of a collaborative effort of an editorial board which includes representatives from numerous security-related organizations such as security tool vendors, academic institutions, and government as well as other prominent security experts.

- **Axelsson's taxonomy of intrusion detection systems**. This project classifies intrusion detection systems according to nine different functionality, structure, and security attributes [13]. These attributes are: (1) detection principle (anomaly, policy, hybrid), (2) detection time (real-time, non-real-time), (3) granularity (batch, continuous), (4) data source (host, network, hybrid), (5) type of response (passive, active), (6) data processing (centralized, distributed), (7) data collection (centralized, distributed), (8) security (low, moderate, high), (9) interoperability (low, moderate, high).

  The major merit of this classification system is that it employs frequently-used concepts that appear in the literature describing security systems and methods. This helps compare intrusion detection tools from the more fundamental perspectives.

- **CIDF: A framework for the cooperation of intrusion detection systems**. The *Common Intrusion Detection Framework* (CIDF) was an effort to develop specifications that

enabled different intrusion detection components to interoperate and share information as well as to allow for the re-usability of modules [196].

The CIDF is based on the assumption that intrusion detection systems (at least the ones developed by projects funded by DARPA) can be modeled through standard components such as sensors, analysis engines, and response elements. The project was motivated by the lack of standardization. The group defined a language to describe objects and relationships through LISP-like expressions that conveyed semantic information. (This language is in fact similar to XML [162] and the proposed structure could be actually implemented with modern ontology-representation languages such as DAML+OIL and OWL).

The project did not have the impact the security community was expecting but it did leave a precedence on how to define and interleave intrusion detection concepts into a single common, meaningful structure. The CIDF motivated the creation of the IDMEF project [43] and the terminology it defined has been used to develop portions of the communication systems of some research projects [112].

- **IDMEF: A standard format for intrusion detection communications**. Motivated by the fact that there is no standardized way for intrusion detection systems to communicate, a group funded by the Internet Engineering Task Force developed the *Intrusion Detection Message Exchange Format* (IDMEF) [43].

  This is a specification for a standard alert format and a standard transport protocol. In fact, it also defines the requirements and structure of intrusion detection systems. These specifications use XML to encode alerts which contain attributes a large subset of intrusion detection products use to report incidents. It defines data types, message classes, alert classes, among other concepts.

  The benefits of using XML certainly give the IDMEF the portability and flexibility intrusion detection systems require to exchange and store critical information, and represents a good way to link intrusion detection tools with other technologies through a standardized language.

  The IDMEF format is fairly new but it has been already implemented in a Snort [170] plug-in to report alarms.

- **A target-centric intrusion detection ontology**. Most terminology used in intrusion detection is described from the point of view of the attacker [103, 126]. Undercoffer and his team developed an ontology to define intrusion detection terms and relationships based on the analysis of over 4,000 classes of computer attacks [203, 202].

  The core structure of the ontology contains types of system targets, mechanism of attack, consequences of attack, and location of the attacker. It is argued that the taxonomic characteristics used to describe attacks should be observable and measurable by the target system. This ontology has been implemented in DAML+OIL and can be used to exchange information between security tools that share the resources this model defines (this ontology is intersected by the outbound intrusion detection ontology proposed in this dissertation; see Chapter 4 for details).

Other related projects have proposed alternate ways to use ontologies in information security problems. Such is the case of an extension to a software engineering ontology that is being

developed to include security considerations while software agents are being defined (like other applications, agent-software is not typically designed with security in mind; [134]). There are new projects in the area of knowledge management and representation that are still in their early stages. They promise to contribute important ontological constructs in the future to the security and intrusion detection areas, as pointed out by Raskin et al. [166].

		Approach		Foundation		Data source		
	Technique name	**AID**	**MID**	**Probabilistic method**	**Learning method**	**Audit data**	**Network data**	**Other**
1	Frequency analysis	X				X	X	X
2	Probabilistic detection	X		X		X	X	
3	Bayesian networks			X	X			
4	Conditional event sequences	X		X		X		
5	Expert systems	X	X			X	X	X
6	Inductive generalization	X		X		X		
7	Neural networks	X			X	X		
8	Attack scenarios		X	X		X		
9	Petri nets		X			X		
10	Genetic programming		X				X	X
11	Embedded sensors	X	X			X	X	
12	Functional dependency	X						X
13	Data mining	X			X	X		
14	SVM's	X			X	X	X	
15	Planning		X			X		
16	Bioinformatics	X				X	X	X

Table 2.3: Classification of intrusion detection techniques

Chapter 3

Problem description

One of the primary topics of this dissertation is a computer security problem derived from intrusion detection for which a multiagent architecture is being proposed. This chapter identifies and defines outbound intrusion detection. It discusses its main features and compares it to intrusion detection techniques making special emphasis on the key advantages this approach has regarding security monitoring. The proposed architecture is described in the next chapter.

3.1 Outbound intrusion detection

At an airport it is critical to verify the identity of those who are going to access the gate area. It should be equally important to make sure no one on board a plane can take control of it with the intention of attacking public buildings or other facilities. Similarly, intrusion detection programs must guarantee that only legitimate traffic and users are let into a system, but it is also necessary to have security measures that prevent users from utilizing the system to compromise others – as typically occurs with DDoS attacks, e.g. [126].

Outbound intrusion detection is a variant of the intrusion detection problem. Its objectives vary from those of traditional intrusion systems but the two approaches share some fundamental aspects such as their monitoring nature and the type of security threats they handle. This section describes the outbound intrusion detection model.

3.1.1 Objective

Outbound intrusion detection (OID) focuses, not on protecting local resources from being compromised as intrusion detection techniques typically do, but on preventing those resources from being used to compromise its neighbors or any other systems connected to the Internet. OID aims at containing the impact of an incident by monitoring outbound activity at the host level so that the system cannot be utilized as an attack launcher or intrusion relayer. Figure 3.1 shows the data collection priority of three different intrusion detection models.

The function of OID is to identify system activity that contains attack signs and which is to be sent through a network channel. The major benefit of this approach is that, unlike incoming traffic whose origin and integrity is difficult to verify, a host is able to tell how outgoing traffic is being generated given the visibility it has of itself. It has access to detailed information such

Sophos Ex. 1007
Page 67 of 216

as the process that generated the traffic, the program attached to that process, the user who started up the process, and even the system calls that were requested to the kernel for execution.

DEFINITION 15

Outbound intrusion detection (OID). Set of techniques used to identify and report attempts to compromise remote systems using network resources and being originated at a local host.

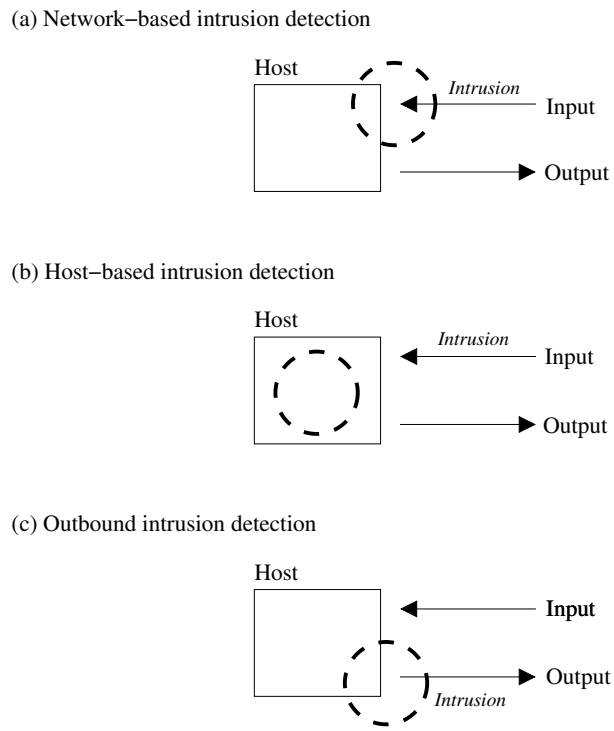


Figure 3.1: Data collection focus of different intrusion detection approaches.

3.1.2 Main characteristics

The OID approach has some distinctive characteristics that are worth exploring from a research perspective. The design of information security systems has been traditionally centered on the protection of critical components and their perimeter [156]. OID allows for the development of alternate security models by moving the spotlight from the target onto the source of an incident. Attack of neighbors is a sign of compromise that can be used to create a security context if the attacks are analyzed at the originating system. The following are some of the most significant features of this method which promotes a less selfish protection mechanism.

1. **A less selfish approach to intrusion detection**. Intrusion detection systems protect resources local to a computer network by monitoring what happens inside the network and by inspecting the traffic addressed to internal hosts. These systems are not concerned about the security of other systems in the outside, and outbound traffic is rarely analyzed. The OID approach aims at developing a net of collective monitoring. The security of a system H is protected by others that make sure no attack is launched from them toward H. H does the same by making sure that no attack is launched locally toward external systems. Although the ultimate purpose is achieving H's own security, each system is not actively pursing its own protection. This creates a monitoring net that provides security to its members that, in turn, would be interested in other systems to become part of the net.

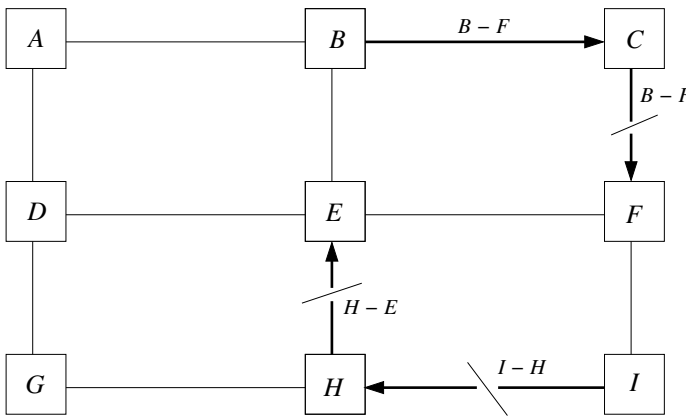


Figure 3.2: Attacks are detected at the originating host or at an intermediate host and can be blocked. This creates a security structure centered on the protection of others.

In Figure 3.5 an attack aimed at H could be detected and blocked by the source system (provided it supports OID). Similarly, if H is being used to compromise E, H itself will detect the incident, block it, and inform E about it. If host B does not support OID and is being used to compromise F through C, the OID system at C could prevent this from happening.

2. **Higher-quality evidence and detection**. "An intrusion detection system can only be as good as the data on which it bases its decisions" [223]. The change in perspective proposed by OID provides increased quality of data.

   A network-based intrusion detection system receives traffic which may or may not come from legitimate hosts or applications – this is left to the security system to decide. The design of the most popular network protocols we use did not take into account some basic security considerations and, as a result, it is possible to modify the fields in a network packet before releasing it into the network. A user can inject packets after having changed, for instance, the source address, the source port number, its network flags, and the segment size in an attempt to evade the security of the remote system he is trying to compromise [161, 143].

The victim system has no way of knowing how these packets were created. It needs to rely exclusively on the information contained in the packets it receives to make decisions, and there are no guarantees about the integrity of these packets – especially if they arrive from a public network like the Internet. An advantage of the OID approach in this respect is broader visibility: a host has total access and control over its own resources.

Some network attacks modify flags and IP addresses on packets as to circumvent the security policy of a firewall [161, 143]. Firewalls that have not been properly configured may be tricked into accepting outside packets they were "supposed" to reject. If the security policy indicates certain packet types are allowed inside the LAN, but they cannot go into the Internet, an attacker could send packets whose source IP address corresponds to an inner host [143]. If no firewall rule verifies that packets were truly generated by internal hosts, the security mechanisms will be bypassed.

A system implementing OID could monitor this sort of traffic by making sure that all outgoing packets have the right source address hereby protecting other systems from being compromised by users at the local host (unlike an outbound firewall that is in charge of checking network packets only, OID complements this kind of checking with the analysis of the process generating such packets in order to detect events more accurately).

The execution of stimulators of intrusion detection systems produces traffic that may contain attack signatures [136]. An intrusion detection tool will react to this input if its detection algorithm is based solely on network packets. OID tools will not react to stimulators. They count on context information in addition to the network packets that enable them to identify what kind of program is being executed. An OID system looks at the process originating the traffic. Traditional intrusion detection tools do not.

3. **Characterization of attack tools**. Signature generation is a task that requires a high level of expertise [126]. In order to extract the essence of an attack, or an attack type, it is necessary to be familiar with the operational details of network protocols, operating systems, applications, as well as the way systems react to certain inputs [142]. Capturing what constitutes irregular activity into a pattern or a signature that can be used by a detection algorithm is not easy. Signatures that try to identify an attack type that covers multiple individual attacks could be too broad and would generate false positives. If signatures are too tight, they will miss small variations of the same attack.

   Given the increasing number of automated attack tools available on the Web [169], it would be desirable to characterize their execution using anomaly-detection techniques like those implemented by other security systems to model the normal behavior of services and applications (Figure 3.3). With the quality and amount of information available locally to OID systems, programs could be characterized and then distinguished from other applications at runtime. Root-kits could be identified, not by the static footprint they leave on a file-system – where they are easily hidden [125]–, but at execution time when they are actually harmful.

   Attack programs that aim at compromising remote systems through the Internet could be modeled through the network traffic they create by taking advantage of the access the host has to the IP stack and kernel data structures of the source system.

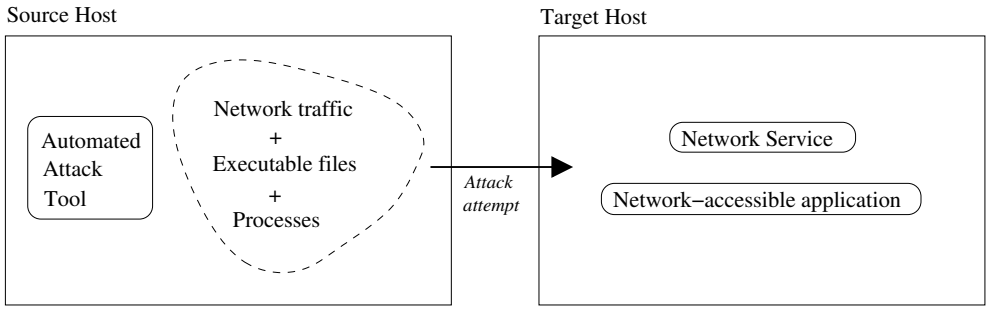


Figure 3.3: Characterization of automated attack tools through the use of locally-measurable and observable variables.

4. **More effective incident response**. Some intrusion detection systems have implemented response to incidents in the past but most of them leave this function to the administrator [126, 13]. If a system is under attack, there are several ways in which to react: firewall tables can be updated to close some ports or to filter out specific packet types; services can be temporarily disabled; passwords can be changed; security patches can be applied; the system can even be brought off-line for post-mortem, forensics work. All these security measures do not prevent the attacking system from targeting a new host should the current victim close all its entry points. This sort of reaction typically helps strengthen the security of the victim system [2] but, besides the effects of the event perceived by the target, there is no easy way to collect further information about the attack, or to react to it by disabling the actual processes responsible for launching the attack at the intruder's machine.

   The type of response an OID-based mechanism can implement is more definitive as access to the source of the incident is possible. There is no need to block local services or ports that could significantly damage the functionality of the system. OID systems are in the position to implement a more desirable type of reaction such as stopping user and system processes, restricting sessions, collecting evidence including the actual attack program and commands used (similar to the information a honey-pot system can capture), and sharing this evidence with the pretended victims so that they are able to heighten their security accordingly. If the system has been rooted, the results of an OID system are uncertain. OID therefore focuses on the majority of users executing attack tools, who are assumed not to have the expertise to modify the system in an attempt to hide their actions.

5. **Early detection of attack and surveillance tools**. Intrusion detection systems raise alarms as soon as they identify attack signs on the inspected evidence. Depending on the type of attack and detection method used, this could take a variable amount of time. An anomaly detector, for instance, could identify in seconds an event that deviates from the security profile of an application, whereas a stateful intrusion detection system could take several minutes to spot evidence of attack on a packet stream.

   By identifying attack tools through signatures, a system may be able to react as soon as the attack program is uploaded into memory, or when it puts the first packets onto the network. The development of response strategies based on an early detection capabilities may allow security systems to limit the extension and consequences of security breaches, not after the

fact, but as soon as the incident arises.

An attack tool may be stored in an encrypted or compressed format (e.g., uuencode, zip, or ciphered with a cryptographic algorithm). This will prevent some security mechanisms that look into the file-system's contents for the presence of well-known attack programs from succeeding. However, attack tools need to adopt executable form in order for them to be run. It is then when they can be more easily identified.

The OID approach supports the idea of detecting automated tools which are used by users to compromise systems connected to the Internet and which account for a significant number of security incidents around the world [124, 169]. It is important to notice that if the OID program is disabled or put to sleep, it will be unable to succeed. That is why external monitoring of OID tools is necessary in order to guarantee they are working as expected.

6. **Improved packet processing without enabling promiscuous mode**. One of the challenges of network intrusion detection systems is the amount of data they need to inspect (see section 2.1.5). These tools operate in promiscuous mode and are generally placed at choke points where they can make the most out of their functionality by looking at a greater percentage of the traffic that goes into the network. However, the high traffic rates of intermediate or interconnection systems force them to drop packets in certain situations. In order to avoid this problem, OID is focused on protecting leaf nodes (e.g., servers and desktop computers) within a LAN by exclusively processing outgoing traffic emanated from them.

   Programs also exist that can detect hosts connected to a segment that are operating in promiscuous mode [2]). They will alert attackers about the potential presence of an intrusion detection system which could make them try to evade detection. An application that does not set a network device in promiscuous mode is more stealthy and may increase the chances of finding illicit activity. Since the objective of OID is to protect systems by observing outgoing traffic only there is no need to enable promiscuous mode.

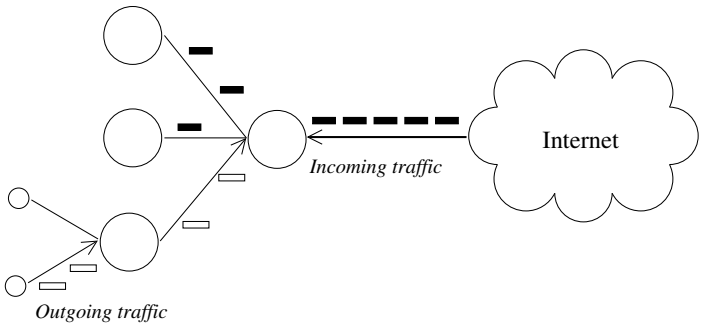


Figure 3.4: Under the assumption that the number of outgoing network packets an internal host generates is less than the incoming network packets a gateway sees, OID allows for more efficient packet processing.

Assuming that the rate of outgoing traffic from a switched-network host is less than the

incoming traffic a gateway or a router receives, this approach allows for better packet handling by bounding the amount of data that needs to be analyzed (Figure 3.4).

OID does not contemplate traffic filtering capabilities as most firewalls and several network-based intrusion detection systems do. These systems identify irregular traffic in order to dynamically block packets preventing them to make it to their ultimate destination. Our approach does not hold a packet and looks into it to decide whether or not it can be allowed to get through. Packets are scrutinized after copying some of them (i.e., the first packets generated by an application) into an analysis buffer and so no packet is retained. This approach does not suffer from the need of having to evaluate the sanity of every single packet, which could constitute a bottleneck problem if the retention-and-checking algorithm is inappropriate for high traffic loads.

OID is a hybrid system that integrates network-based and host-based data sources. It distributes the detection task among all the hosts of a LAN instead of centralizing it on a firewall or router. Every system is responsible for monitoring its own outbound traffic, which constitutes a more scalable processing strategy.

7. **No need to keep stateful transfers for the reassembly of fragmented packets**. Many network-based intrusion detection systems work with complete packets only and do not provide reassembly support for fragmented packets [90, 143]. Since network-based intrusion detection systems need to evaluate packets as quickly as possible as to avoid flow congestion, some of them do not keep track of the fragmentation state of packet streams due to the overhead generated by having to delay a packet's evaluation until all its fragments are put together so that the detection algorithm can be applied. Evasion exploits based on fragmentation split a packet into pieces that individually bypass firewall controls and other filtering mechanisms hence achieving undesirable results on the target [143].

   OID handles packet reconstruction differently. Since it has access to complete packets at the sending location via the IP stack, packets can be analyzed before reaching the fragmentation phase – in case they have a fragmentation flag turned on. The reassembly overhead does not appear when capturing packets at this point of the transmission. This significantly expands detection coverage to include local attack attempts that may circumvent the security mechanisms of a remote system by dividing the payload of a packet through the use fragmentation. An attack could generate raw packets to evade OID but, since the OID tool is not only looking at network traffic but also at the actual processes in execution, detection is still possible.

8. **Possibility of detecting attacks over encrypted channels**. A major disadvantage of intrusion detection based on the analysis of network traffic is encryption. An encrypted data transfer will render a detection system ineffective if it captures packets and applies matching algorithms to the payload portion. Cryptography effectively scrambles data to protect its confidentiality and integrity [178]. Frequency analysis and many other cryptanalysis techniques are not effective with strong cryptographic algorithms which could allow an attacker to hide information detectable by an intrusion detection system. The use of encryption in virtual private networks (VPN's), network protocols, email applications, as well as terminal

emulators and network-connection software, generates packets that cannot be handled by intrusion detection programs.

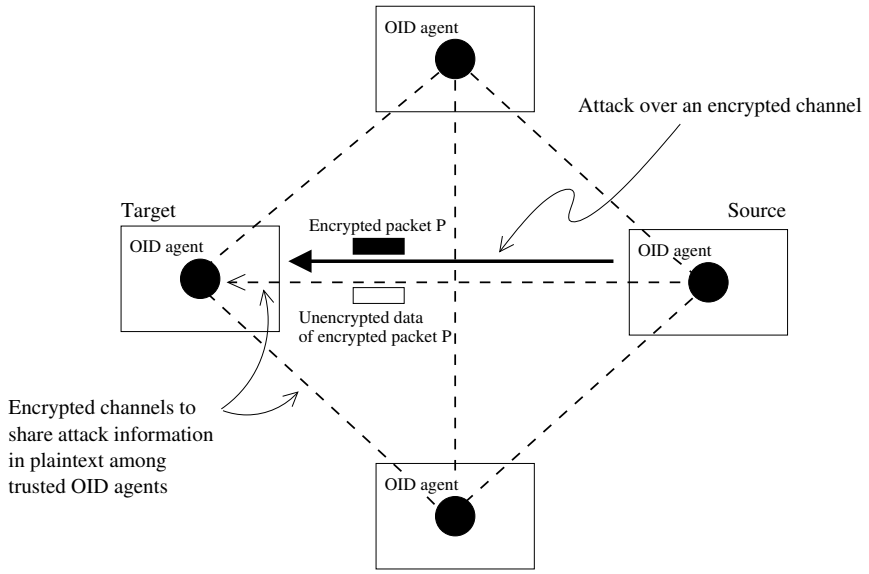


Figure 3.5: A mesh of trusted OID agents that share plaintext of encrypted attack data over secure channels

With OID it is potentially possible to look at certain data before it is actually encrypted. For instance, it can be determined whether an encrypted session contains commands crafted to defeat a target system. That is because the commands can be looked into in the clear by capturing the command history of the user. A mesh of trusted, certified intrusion detection agents could be created to alert systems about the presence of attackers like these who are using encrypted sessions (Figure 3.5; if the information is encrypted at the application level, this is not possible). A network-based intrusion detection system installed at the target machine is blind to this kind of traffic, but can receive information from an OID system at the attacking location which has access to the execution details of the attacker's processes, including plaintext.

9. **Improved chances to detect stealthy attacks expanding over long periods of time**. "Intrusions require the accumulation of substantial state information for their identification" [2]. An effective way to escape detection by some intrusion detection systems is to extend the attack period by inserting delays between subsequent actions. These delays will give time for the system to receive other network traffic and, given the cost of maintaining an intrusion's state, the detection system may drop a pending analysis line if the next set of actions does not arrive before a certain threshold [126]. In an attack modeled by a state-machine or a Petri net, for instance, the initial transitions leading toward a compromise state may quickly appear, but the detection process might not complete if future actions take sufficient time to appear as to exceed the intrusion detection's threshold.

If this sort of attack is embedded into an attack tool, an OID system at the source location could identify it at execution time. System-call traces, function-call dependencies, as well as resource usage are distinctive features of a program that can be used to identify them. This can help defeat attacks that expand its activity over several minutes or hours.

10. **Performance improvements over time by signing safe programs**. Intrusion detection systems continuously monitor network and host activity. They either compare the data they observe against normality profiles, or against attack signatures.

    Similarly, an OID system can check the execution of programs against a data structure containing attack-program signatures. Once a program has been verified, it can be signed as to indicate it is safe to execute. It will not be necessary to verify it again unless new signatures are added or unless its integrity has been compromised (remember that a program is deemed as dangerous only if it matches a network and/or a process signature; programs for which no signature exists will go unnoticed; the halting problem prevents a security system from determining whether any given program will produce risky results). In the long run, this will benefit the performance of the OID security mechanism which will have to run only when new programs or when new program signatures appear (Figure 3.6).

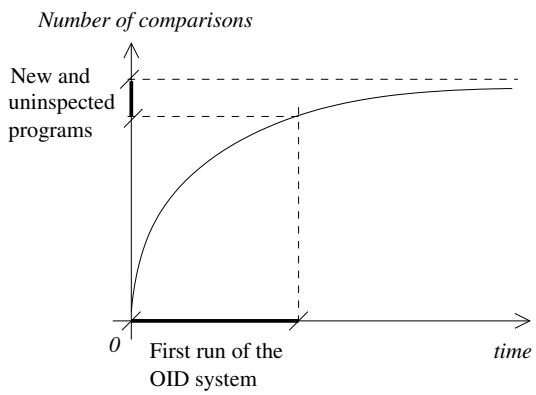


Figure 3.6: Performance of an OID system: *Number of comparisons* **vs.** *time* **(a comparison tries to match program files against program signatures).**

11. **Resilience to denial of service attempts**. If an OID system was protecting a host, a user might try to launch a denial of service attack in order to bring it down locally. Having an OID access to the source of such attack (it executes on the same machine as the attack program) it is able, not only to realize it is under attack, but also to react to the incident by suspending sessions or any other measures to make the threat disappear. This makes of this approach a more robust security monitoring strategy.

    Additionally, distributed denial of service attacks are usually executed using remotely managed tools. At one of the compromised servers on which an instance of the attack tool has been downloaded, an OID program could be executing. If a signature exist for the tool,

the security system will detect it and will be able to alert the remote target(s) about the existence of a latent threat.

Chapter 4

A multiagent outbound intrusion detection architecture

This chapter describes a multiagent architecture proposed for the problem of outbound intrusion detection. It discusses main tasks and agent classes, the characteristics of the environment, introduces the underlying ontology that allows the interaction of agents and describes the functioning of the architecture.

4.1 A multiagent approach to intrusion detection

Given the very specific tasks that need to be developed by an OID system, the information exchange necessary to integrate these tasks, and the intelligent functions involved in intrusion detection, the multiagent approach appears as a good option.

Intrusion detection is a process requiring human skills that researchers have tried to build into computer programs [65]. An administrator needs to analyze different sources of dynamic data (e.g., kernel data structures, audit log files, and network traffic) which may contain a few, small pieces representing security information. In order to do that, he will need to correlate events very precisely based on attack patterns he knows, or by identifying and interpreting irregular signs during the operation of the system [67]. If any security incident has occurred in the past, the administrator will use that previous experience to handle the data and detect attack attempts more accurately.

This kind of monitoring not aided with intrusion detection tools could be performed in real time for a single host or a small network. Monitoring a WAN or a large corporate LAN would definitely be a tough task for a single administrator or a small team of security analysts given the dimensions, dynamics, and heterogeneity of computing platforms used nowadays [101]. Training a security team or hiring remote administration is an expensive solution but, even when security issues can arise in any kind and size of organization, the costs can prevent the management team from investing in the protection of its information assets. Intrusion detection tools must be used to automate vigilance, reduce costs, and increase effectiveness. The development of intelligent software which helps simplify systems administration is necessary.

Agent-based architectures are an option to implement the functions needed to perform in-

Sophos Ex. 1007
Page 77 of 216

trusion detection [224, 15, 223]. These functions can be assigned to specific agents that will be responsible for performing certain tasks autonomously. In order to enable inference processes, meaningful communication, and smooth integration with other security mechanisms, it is necessary for agents to have a common understanding of the different objects and relations present in their environment. This can be achieved with ontology support as mentioned later in this chapter.

4.2 Specific objective of the architecture

The two main approaches to intrusion detection are to model what is normal, and to model what is abnormal. The creation of normality profiles that describe licit and regular operations is not an easy goal to accomplish [108]. The number of kernel components, devices, and applications that compose a system, along with the execution states, variables, events, and messages it handles constitute a complex model. A system like this can be described, but the task of characterizing it through the definition of a sense of "system self" which captures stability and safety conditions has proved challenging [83, 64].

A simpler problem is to characterize what constitutes an abnormality – assuming that the majority of the events within a system are normal [64]. In the case of OID, characterizing outbound activity, such as network traffic, poses similar problems to the ones that appear when modeling the normal functioning of a system.

"Regular traffic is not well-behaved" [143]. Due to the number and diversity of the systems that can connect to the Internet as well as the number of protocols they utilize, it is not trivial to generalize what normal traffic is to a router or a firewall. Creating behavior profiles of the outgoing traffic of a host might seem a simpler issue, however, the execution conditions of internal applications as well as the service requests that need to be processed, generate packet streams for which it is be difficult to define a single, precise normality profile [143].

Our particular approach to OID will be to define signatures of illicit activity. We leverage the quality of information available to the host that describes its internal structures and execution state in order to precisely detect patterns that may indicate someone is trying to compromise the system by launching attacks from a local host.

Given the trends observed in information security over the last years (Section 2.2), we will concentrate on the more specific problem of *characterizing remote attack tools that generate IP packets detecting them at runtime and reacting to these incidents in an automated way.* IP is chosen because it is the most widely used protocol on the Internet and it is also frequently used to perpetrate cyber-attacks [142, 161]. The detection mechanism will be connected to response methods that will counter compromise attempts locally, will alert neighbor systems, and will trace the intruder back in case he is connecting through an interactive shell session to the local server [118].

4.3 Characterization of automated attack tools

OID's objective is unselfish: to achieve the system's own security through the protection of others [119]. If the system watches for the security of its neighbors, and these neighbors follow

the same approach, a higher, cooperative security level will eventually appear.

One way of protecting other systems is to identify users who may be trying to bring them down. A common way to analyze and attack network servers and workstations is through the use of automated attack tools [126]. Users download these programs from the Internet with the intention of compromising all sorts of systems – although sometimes they just want to play a prank on someone's workstation.

In order to identify these tools, one could just look for the static file image of the program as integrity checkers and root-kit detectors normally do [100]. Unfortunately, compression and encryption make this approach unfeasible. The program, however, will have to adopt executable form so that it can be run. It is then when attack tools can be detected at the hosting system and administrators can react to the incident more definitely. They can also warn the attempted victim about the possibility of being the target of a wider, orchestrated attack (e.g., distributed denial of service).

Many aspects may be considered for the characterization of programs [184]. We consider two groups of them: (1) the peculiarities of the network traffic generated by an attack tool, and (2) the resources accessed by its process(es) when in execution (note that the OID architecture presented in this document concentrates on remote attack tools only). The former is an approach widely used by network-based intrusion detection tools. The difference is that an OID system has access to details such as how packets are being actually generated (including how the source IP address and/or port is being spoofed) and this information is not accessible to traditional intrusion detection systems which are located on the victim's side.

Analyzing the resources accessed by an attack tool also helps identify its execution. This is because features like the data and text segments uploaded into RAM, the libraries and other files open by the program, as well as the sequence of system calls requested to the kernel are, as a whole, distinctive enough to identify a program from the rest [184]. Similarly, this type of attack software frequently crafts network packets with very distinctive characteristics that can be easily identified as anomalous or intentionally-wrong. If packets that belong to the same exchange are captured and put together for analysis, it may be possible to identify the attack program that originated them. This is one of the hypothesis of this project.

4.3.1 Attack program signatures

In order to capture execution and communication data to model a program, two signature types are considered. They are used independently by detection agents which in turn supply their matching results to correlators that generate a composed diagnosis. Each signature should contain attributes that help uniquely identify an attack program.

1. **Network traffic signature**. An OID traffic signature must contain information extracted from the sequence of network packets generated by an automated attack tool. The TCP/IP traffic signature presented here considers the first k packets of an exchange (outbound packets only). They are captured and condensed in order to build a single signature that describes them all. Some attributes are more distinctive than others for purposes of packet stream analysis [143]. The proposed structure is as follows:

```
{Source Port, Source Address, Destination Port, Destination Address,
 Fragmentation, Window Size, {Flags 1, Flags 2, ..., Flags_ k}, {Hash 1,
```

```
Hash 2, ..., Hash k}}
```

`Source Port` data is usually modified in evasion attacks in order to circumvent filtering rules. Clients typically use non-privileged ports to contact a server listening on privileged ports; although some servers contact other servers using privileged ports, as in the case of DNS's port 53, seeing packets with a privileged port number as source port going out of a workstation may constitute a sign of anomalous activity.

The `Source Address` field may also suffer from the same type of problems as source port numbers do. An attacker may be inserting addresses of the internal hosts of a network in order to bypass a misconfigured bastion firewall that accepts incoming packets from the outside as long as they have a local source IP address.

`Destination Port` and `Destination Address` can be both used to characterize tools that aim at a particular target or service.

`Fragmentation` flags may indicate the attacker is forcing packet fragmentation in order to avoid network-based intrusion detection systems that look for suspicious strings in the payload but which do not support the expensive task of packet reassembly. (Notice that this "intention" information is rarely accessed by intrusion detection systems which only see packets but have no way to tell how they are being generated).

`Window Size` is an attribute used in surveillance operations to fingerprint an operating system by seeing how it reacts to malformed packets (e.g., those with a '0' window size; different protocol implementations react differently to exception and error conditions and this can be used to identify the type or version of the services a host is running).

TCP `Flags` are one of the most evident signs of furtive and malicious activity. They can be altered in order to influence the synchronization of an exchange (e.g., to do session hijacking [143]) or to launch denial of service attacks (e.g., a SYN-flood attempt).

The above fields can be used to distinguish abnormal sessions in an anomaly-based fashion. An additional set of attributes that can be used to more accurately identify the originating program is packet payloads. Many attack tools use root-shell code[1] which represents a very distinctive characteristic of a program and its programmer [211].

Other fields such as packet checksums, time-to-live values, and sequence numbers [142] (Appendix B provides a description of the TCP and IP packet formats) can be also considered but we decided to include some of the most descritive ones only.

2. **Process signature**. Profiling a program in execution involves aspects which are operating-system dependent or that depend on the peculiarities of the executing environment (e.g., configuration and set-up conditions caused by the software being executed [184]). Some of the attributes that a process signature may contain are captured in the following signature structure:

```
{Program Name, Execution Parameters, Text Segment Checksum,
 Number of Files Open, {File Open 1, File Open 2, ..., File Open k},
 Number of Dynamic Libraries Open, {DLibrary 1, DLibrary 2, ..., DLibrary k},
 Memory Allocated at Upload, Hash of the File Image}
```

[1]Root-shell. A sequence of commands precompiled for a particular architecture and stored as a character array on an attack program that will be inserted into memory during a buffer-overflow attempt

With a different weight, these attributes will build a platform-dependent signature that describes part of the behavior of attack tools at runtime.

Unlike the overall structure of a program, `Program Name` is an attribute easily modifiable. However, some attackers may download and execute the tool as-is and file names would help identify the type program being used.

`Execution Parameters` are particular to each program. Sometimes the options passed in the command line need to follow a fixed order, and options are typically limited and very specific.

The `Text Segment` contains code that will be executed by the processor. It is dependent on the source code and the compiler used and reflects the way the program has been structured. A representation of the segment (for a determined architecture) can be used. (In the case of scripts, this attribute is not being considered as it is not straightforward to extract the code in executable format from a file image. The actual code of the script is used instead).

The `Number of Files Open` is a characteristic of a program that can be used to complement the other attributes without having to do process tracing to monitor system calls. The specific names of these files (`File Open` i) as well as the order in which they are open can also contribute to the description of programs in execution.

Many programs link `Dynamic Libraries` made available by the operating system. The number and name of these libraries, as well as the order in which they are accessed is information that reveals the internals of the monitored program.

The amount of RAM `Memory Allocated at Upload` time, including text and data segments, can also be captured as a descriptive attribute.

The `File Image` stored on hard disk is a feature used by integrity checkers in order to detect illicit modifications. Compilation, compression, and encryption can render this attribute invalid, however, using the right weights, it can be helpful in the detection of tools that have been downloaded and executed without being prior modification.

4.4 Environment

The environment OID agents inhabit is a network host. All observable and measurable information within the host constitutes a potential perception to the sensing agents of a multiagent architecture. Agents that implement incident response will in turn modify this environment based upon security-relevant events about which they have been warned. This environment can be described according to different *dimensions* that help understand it and determine the appropriate agent architecture to use [173] (Figure 4.1).

By the capability of an agent to acquire meaningful information, the environment is *fully observable*. The OID approach allows agents to access complete state of the environment. The information captured by the sensing tasks is enough to decide what to do at any given time.

By the sequence of events observed by an agent, this environment is non-deterministic or *stochastic*. An action performed by an agent cannot always have the same outcome. Random factors affect the result of an action such as a change in a program's state or an error condition. The outcome of an OID agent action can be any of a set of possible results whose probability is variable (a response agent, for instance, could have sent a termination signal to a process but a parent-child binding could have prevented this from happening as the agent was expecting). Furthermore, there are elements whose behavior cannot be known a-priori such as users and

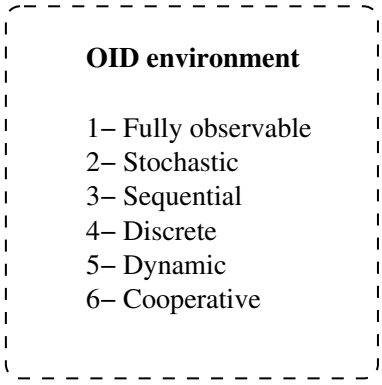


Figure 4.1: Features of the OID environment

the reception of network packets from clients. These elements have also a direct impact on the result of actions and the status of the system.

This environment is *sequential* as the choice of actions by an agent at a certain point in time depends not only on the current state but also on previous events that influenced the current state. The detection of intrusion actions is highly dependent upon events observed in the past which helps interpret the contents of messages at present. Additionally, an action performed by an agent can and will affect other actions and decisions at subsequent times. The state of an intrusion event needs to be followed in order to determine the point at which illegal actions are being performed so that evidence can be obtained.

The host environment is *discrete*. The number of percepts types and actions relevant to OID agents is limited. The possible execution states of an application form a finite set as well – error conditions can be many, but finite at the end–, and the host's timeline upon which events appear is discrete too.

The OID environment is *dynamic* since it can change while agents are making decisions. This dynamics is part of any multiuser or multitask computer environment which adds difficulty to the modeling of its components and their interaction, the inspection of data streams, and the reaction incidents.

The approach to OID we are taking defines a *cooperative multiagent environment* in which a number of diverse agents contribute to the goal of identifying attack programs in execution. All the agent subgoals are defined toward the achievement of that common objective and no competitive function is part of the approach.

4.5 Task definitions

The task of characterizing, detecting, and reacting to automated attack and surveillance programs constitutes a phase of misuse detection that integrates direct monitoring of host-based data sources.

From the literature on agent-based intrusion detection systems, there are three high-level tasks common to all of them [92, 41, 224]: (1) data collection, (2) data integration, (3) data correlation. In addition to these, signature deployment, evidence sharing, and other functions

are also implemented.

In the case of OID, we can identify four high-level tasks very similar to the ones mentioned above which are composed of a number of more specific, atomic sub-tasks. The four major tasks are: (1) sensing, (2) signature generation, (3) signature detection, and (4) reaction or response.

1. **Sensing**. Intrusion detection systems model rogue processes and attacks by using different variables such as system-calls invoked by a process (sometimes including parameters) [83, 211], the set events generated by a process [108, 109], modifications made to kernel data structures [9], the functional structure of programs [186, 211], and the impact a program has on resources usage when it is executed [50, 5].

   Data collection, also known as *sensing* when discussing agent systems, is a basic function needed to obtain all that information. Data must be screened in order to optimize communications with subsequent phases. There are two subgoals of data collection for OID: (1) to observe an attack tool in execution in order to extract data to be used in signature generation, and (2) to monitor the system during regular operation in order to supply the detection task with analyzable data.

   The better the quality of the data (e.g., complete, compact, trustworthy, properly-formatted data), the faster the matching may be and the better the chances of providing a suitable security level to the resources being protected.

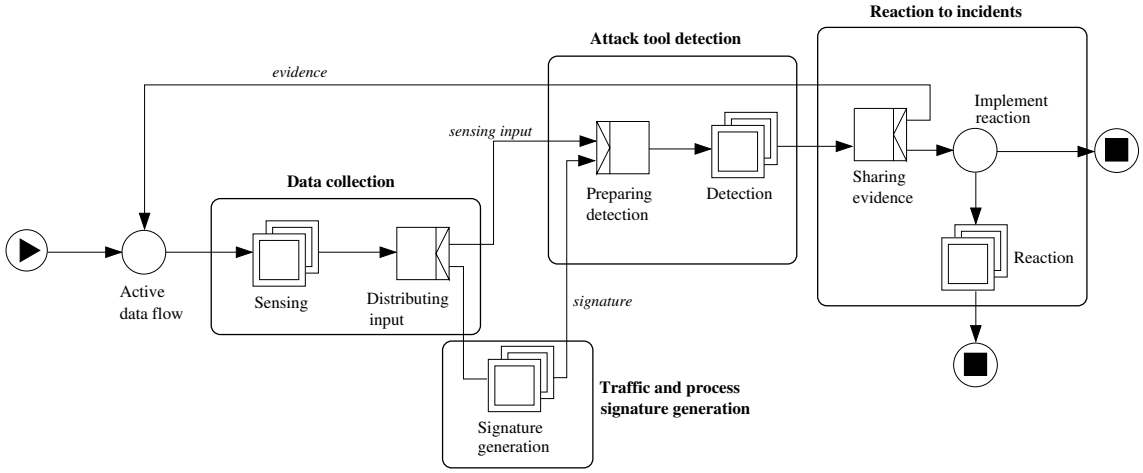


Figure 4.2: Primary OID tasks (YAWL diagram; see Appendix A).

2. **Signature generation**. The functions of signature generation are to integrate data emanated from the program to model, and to condense it into detectable patterns. The critical component of an automatic signature generation algorithm is to determine which aspects of a program need to be taken into account. These have to be distinctive enough for distinguishing a program from the rest and, at the same time, flexible enough to effectively detect small variations of the program.

Signature design also needs to consider the suitability of the signature format being used [142, 43]. Signatures are supplied to the detection module and, if the format has not been properly designed, or if the signature contains redundant or irrelevant information, detection will be unnecessarily delayed.

The signature generation task is also in charge of accumulating all the signatures on a repository that will be accessed by the detection mechanism at execution time, and it will also update the repository as new, better, or incorrect signatures are reported.

3. **Signature detection**. The detection task is a data correlation function. Information received from the sensing modules needs to be integrated and interpreted in order to tell whether or not there is something anomalous in the system. For that purpose, the signature repository needs to be open and all pertinent signatures retrieved. Once system activity data has been properly structured and the signature bank uploaded into memory, the detection algorithm can be applied.

   Given that the OID focus of this architecture is on the detection of attack tools used to compromise remote systems through network channels, the event that will trigger the signature detection algorithm is the request to access a network device for initiating a data transfer (e.g., a *socket()* system-call requested to the kernel). When this sort of event is detected, it will be necessary to inspect the execution of the program in order to determine whether it matches or not any of the signatures.

4. **Response**. Automated response relies on accurate detection [126, 60]. There are two types of response: (1) passive and (2) active. Passive response to incidents includes session logging, increased monitoring of a service or process, and sending alert messages to the administrator. Active response has higher detection-accuracy requirements as it reacts to incidents by drastically modifying the environment with the intention of containing attack attempts.

   The consequences of inappropriate response to misdiagnosis can be severe especially if reconfiguration is being performed, processes are being suspended or killed, sessions are being terminated, files are being deleted, access is being denied, or if an emergency shutdown is scheduled [2].

   An OID system can effectively perform both types. Through active reaction it could, for example, suspend a hostile process in execution, and through passive actions it could alert neighbor systems about the existence of a security threat.

4.6 Task-to-agent mapping

The four high-level tasks of OID can be mapped onto specific agents that will implement, along with other support-function agents, the mechanisms to monitor the execution of programs. Tables 4.1 and 4.2 describe the three main agent classes that compose this architecture, namely, *Sensors, Correlators*, and *Reactors* (agent subclasses corresponding to each of these major classes are defined by the ontology; see Section 4.10).

Class name	**Sensor (S agent)**
Purpose	To provide data collection capabilities for the multiagent architecture. This agent class is in charge of monitoring data sources and to extract meaningful information from them that serves the purpose of generating and detecting signatures.
Perceptions	(1) Data streams observable within the host (e.g., environment variables, network packets, system-call sequences) (2) Messages from other agents
Actions	(1) To listen for system activity that indicates signs of intrusion attempt (2) To screen data based on the relevance of the information it carries (3) To decide what information to send to peer agents (4) To send data to other agents

Class name	**Correlator (C agent)**
Purpose	(1) To integrate data from multiple sources in order to create program signatures (2) To integrate data from multiple sources in order to detect program signatures (3) To trigger the response mechanism by alerting reactor agents about intrusion events
Perceptions	Messages from other agents
Actions	(1) To create program signatures (2) To store signatures on the repository (3) To detect program signatures

Class name	**Reactor (R agent)**
Purpose	To implement passive and active response to security incidents as identified by correlators
Perceptions	(1) Messages from other agents (2) Status of system elements related to incident response (e.g., files opened by a process, process status) (3) Status of security mechanisms related to incident response (e.g., active firewall rules, file permission masks)
Actions	(1) To decide on the response type to implement according to the detected event (2) To develop active and/or passive response to incidents (3) To store evidence on the repositories

Table 4.1: Description of the Sensor, Correlator, and Reactor agent classes

Class name	**Name and Ontology server (NOS agent)**
Purpose	(1) To keep track of the location of agents (2) To supply location information to agents (3) To update the ontology image (4) To supply the ontology definition to new agents (5) To report changes on the ontology to all agents
Perceptions	(1) Creation of new agents (2) Destruction of existing agents (3) Changes in the status or location of agents (4) Ontology update events (5) Information request messages
Actions	(1) To register agents (2) To remove agents (3) To update agents location and status (4) To supply location information (5) To update the ontology (6) To send ontology updates to all agents

Table 4.2: Description of the Name and Ontology Server agent class

4.7 Organization of single agents

All the architectures described in the intrusion detection literature employ one or more processing modules – called agents by some implementations – arranged in many different ways. Seven architectures were analyzed in order to identify the most common inter-agent connections ([92, 224, 15, 25, 41, 78, 221, 158]). As mentioned earlier, these architectures have some functional commonalities that make them comparable.

Taking three basic agent classes described in the previous section, we describe four connection architectures and discuss their advantages and disadvantages (agents S, C, and R agents are used; since no architecture uses an ontology and only two architectures give notion of how agents are located by others [92, 221], the NOS name and ontology agent-type is not included). The R reactor agent in this analysis is responsible for any activity suggested by correlators as a result of data analysis.

Figure 4.3 depicts the four models. Both, peer-to-peer and tree structures were considered. We focus on the connection of the agents to the data sources and on how they exchange information.

1. **Universal-sensor architecture**. This is a simplistic architecture that could be considered a monolithic structure from the functionality perspective. It has a hierarchical structure containing a sensor, a correlator, and a reactor. The sensor is in charge of monitoring all the

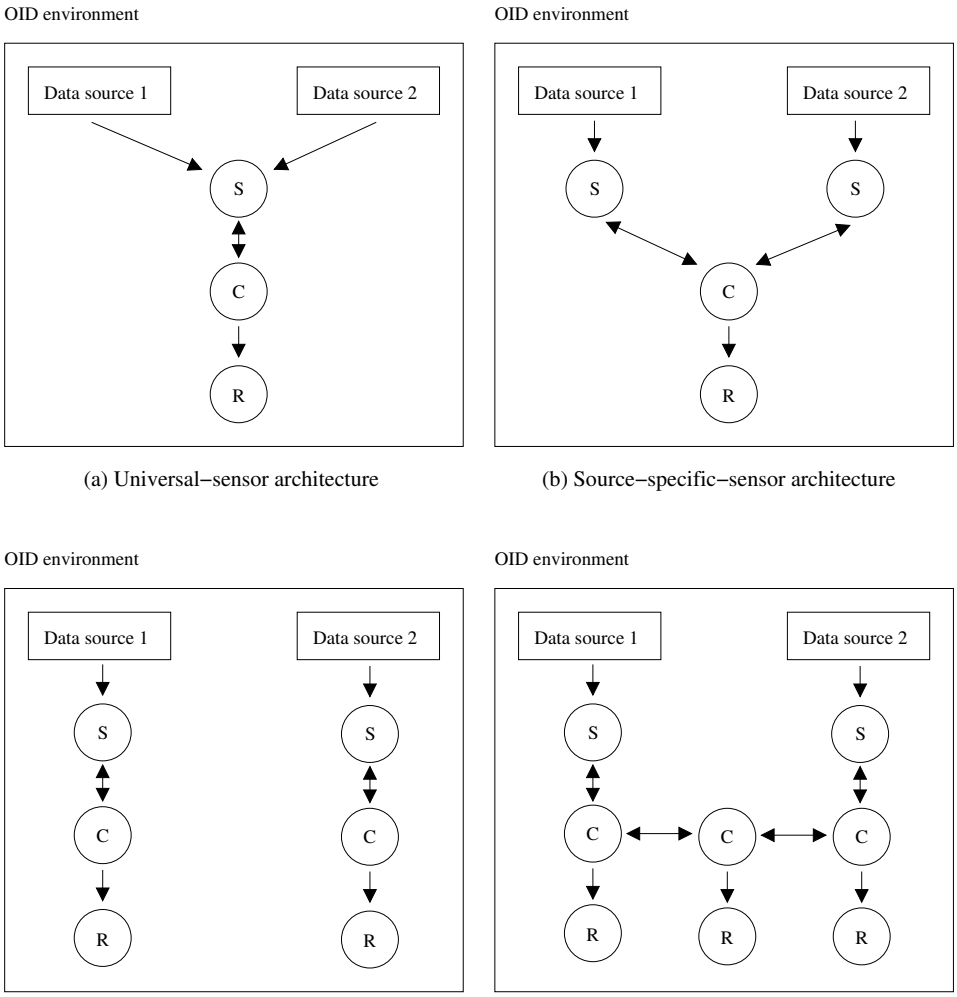


Figure 4.3: Agent interconnection architectures (S = sensor, C = correlator, R = reactor; arrows indicate data exchange flow).

available data sources. There is a unique correlator that generates and detects signatures, and one reactor that takes care of incident response.

– *Advantages.* It is a model simple to implement and the consumption of resources is kept to a minimum.

– *Disadvantages.* Integrating the functionality to access different data sources into a single agent might not be the best alternative. It may increase complexity unnecessarily as the agent should carry the code to handle all data sources available. This model limits portability and does not scale well to the addition of more data sources (basically, the agent needs to be changed). A major disadvantage of this hierarchical structure is that, if the sensor agent is not working properly or not working at all, the whole mechanism gets stuck.

2. **Source-specific-sensor architecture**. This architecture has independent, specialized sen-

sors for each data source. The sensors send the data to a central correlation agent that triggers incident response by contacting a single reactor.

– *Advantages.* The task of monitoring data sources is split into several sensors. This allows for dynamically adding and removing sensors depending on the need to monitor more data sources at the same time. The correlation mechanism still receives evidence from different data sources which increases the likelihood of identifying illicit programs. If one of the sensors fails, other sensors are still able to provide the correlator with data and to detect intrusion attempts.

– *Disadvantages.* In case the correlator gets disabled or presents any functional problem, there is no other agent with the capability of detecting and generating signatures. The system, in that case, would fail and no reaction would be carried out.

3. **Parallel architecture with no correlation**. This model includes independent, parallel mechanisms for detecting intrusions on each data source..

   – *Advantages.*It is a more robust model in terms availability. If one or more of the agents in charge of monitoring a certain data source goes off, the system is still able to provide protection – including reaction to incidents – for one portion of the system.

   – *Disadvantages.* Intrusion detection systems that integrate multiple data sources have proved more effective at detecting attacks [1]. The main weakness of this architecture is that it does not take advantage of the availability of multiple data sources. This might increase the rate of false alarms as correlators are not working with as much information as if they were receiving data from all evidence sources.

4. **Parallel correlation architecture**. This architecture integrates the best features of the other architectures presented before. There are independent agents in charge of the monitoring of each data source and they supply, at the same time, evidence to a central correlator that fuses the data and has the potential to identify intrusion attempts more accurately.

   – *Advantages.* This system has a central correlator that receives and fuses information from different sensors in order to determine the best way to react to incidents. In case this correlator fails, the system is still able to provide full protection by detecting attack signatures at every data source. In the event a sensor fails, the rest of the mechanism is still operational and keeps providing security monitoring to the system. Since the sensing functionality is stored on independent sensors, this model allows for portability and scalability.

   – *Disadvantages.* A disadvantage of this architecture is the number of agents it requires. The first architecture (see (a) in Figure 4.3) used three agents whereas this architecture uses $3n + 2$ agents, with n being the number of data sources to monitor. This is computationally much more expensive if agents execute separately – e.g., agents are stand-alone processes; it is a different story for agents implemented as threads [110].

Out of the four models, the last one presents clear advantages regarding availability. The isolation provided by grouping agents by data source and then connecting these "stand-alone" groups through correlation elements increases the robustness of the structure while still providing evidence fusion.

This agent groupings are exploited in the proposed OID architecture described next in order to develop a model more resilient to the malicious-host problem[2].

4.8 Agent cells

The proposed architecture is not composed only by agents that operate individually and co-operate with their peers [224, 41]. We group agents functionally into teams that cohabit sub-environments called **agent cells** (Figure 4.4).

DEFINITION 16

Agent cell. A functional software construct that groups several agents for the purpose of achieving a common goal.

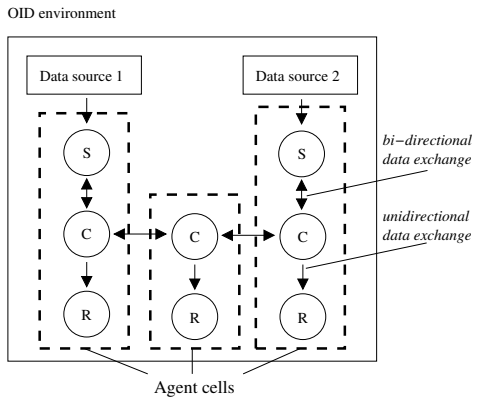


Figure 4.4: Grouping agents functionally into specific-purpose sub-environments or *agent cells*.

An *agent cell* is a software environment which agents can trust. Unlike a host, whose overall hardware/software integrity can be difficult to asses, virtual machines and interpreters are software constructs whose operational safety is easier to verify mechanically by the agent themselves through digital signatures or other cryptographic means. An agent implemented as a process would execute on the operating system directly [15, 158], but a cell agent executes inside the cell which is the one in touch with the underlying operating system. The task of securing execution is left to the cell and not to each of the agents.

Agents that share a common goal are put into the sub-environment of the same cell. The tasks necessary to develop intrusion detection are actually implemented by the agents but cells integrate them functionally for security and communication purposes. Figure 4.5 shows the agent cells that compose the architecture.

[2]Malicious-host problem. The execution of agents on a rogue environment that gives no guarantees regarding execution correctness or agent integrity [153, 178]

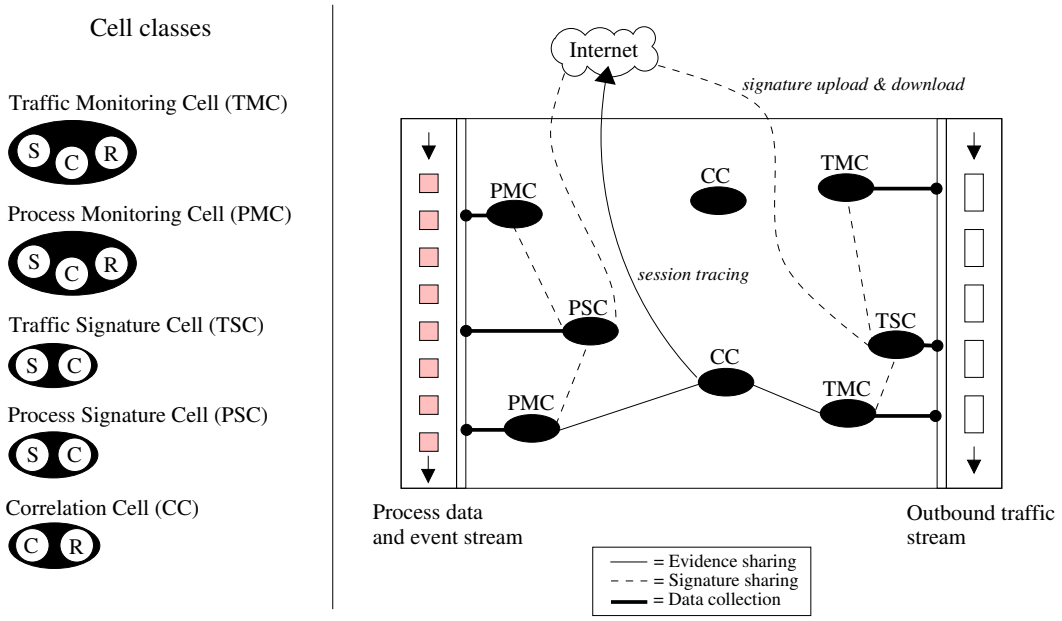


Figure 4.5: OID architecture and main cell classes.

The creation of cells allows agents to perform independent detection functions. A cell can be focused on network-level intrusion detection while another one works on process-level detection. Then, both can be integrated by means of correlation agents. Cells are self-contained in the sense that sensing, detection, and reaction capabilities are all available inside the cell. Agent cells are arranged non-hierarchically as a way to provide increased robustness to the OID system (Janakiraman et al. study peer-to-peer intrusion detection in [92]). If one of the cells is down, others will be available.

In Figure 4.5 for instance, if a TMC cell is down (see cell-type descriptions below), another TMC cell or a PMC could be up, and vice versa. Similarly, if a CC correlation cell is down, the rest of the monitoring cells will continue to provide protection to the platform even when the correlation functionality may be off.

4.8.1 Agent classes and cell classes

There are several agent and agent-cell classes composing the architecture. Agent types were introduced before and here we describe their sub-classes (for an overall description of the classes involved in the architecture, see the ontology in Figure 4.12 later in this chapter):

- *Sensors*. Sensors are responsible for collecting data. They are further classified into *Traffic Sensors* and *Process Sensors* depending on the data source they access.

- *Reactors*. Reactors implement reaction procedures to incidents. They are further classified into *Guards*, which execute local response, and *Tracers*, which connect with other agents located at remote hosts in order to back-trace suspicious sessions identified in a security incident [118].

- *Correlators*. Correlators receive data from *Sensors* and inspect it through detection mechanisms. They are also responsible for triggering response from *Reactors*. Correlators are further classified into *Traffic Detectors*, *Process Detectors*, and *Hybrid Detectors* – hybrid detectors that fuse evidence from the other two detector types.

A number of instances of these classes are the ones performing signature generation and detection, however, in order to achieve higher-level goals such as implementing network-based OID, individual agents are put into agent cells, which can be of five different types:

1. *Traffic Signature Cell (TSC)*. Composed of agents S and C. S provides traffic data to the correlator C (a *Signer*) in order to generate traffic signatures corresponding to the input attack tool (Figure 4.6).

2. *Process Signature Cell (PSC)*. Composed of agents S and C (notice that agent types are generic; specific sensor and correlator sub-types exist for the tasks of handling process and network data; see ontology description in the next section). S provides process data to the correlator C (a *Signer*) in order to generate a process signature corresponding to a certain attack tool (Figure 4.6).

3. *Traffic Monitoring Cell (TMC)*. Composed of agents S, C, and R. It implements attack tool detection by processing network packets. S provides traffic data to the correlator C (a *Detector*) which in turn may command R to react upon an event (Figure 4.7).

4. *Process Monitoring Cell (PMC)*. Composed of agents S, C, and R. It implements attack tool detection by inspecting process data. S provides process data to the correlator C a *Detector* which in turn may command R to react upon an event (Figure 4.7).

5. *Correlation Cell*. Composed of agents C and R. It combines the output received from traffic and process monitoring cells (TMC and PMC) in order to more accurately detect the execution of a rogue program. Once C receives information from monitoring cells, it may command R to react upon an incident (Figure 4.8).

4.8.2 Signature and ontology deployment

There are two signature-related processes that compose the architecture, one to generate signatures, and one to match them against observed events. Signature generators are called upon by a user who wishes to generate signatures for an attack tool. These signatures will be utilized by the monitoring agents in order to automatically detect attack attempts launched by a local (legitimate or intruding) user.

Signatures can be generated and used by local agents only, however, they can also be shared with other systems. Alternatively, they can be generated at a central location and then deployed to agents. Many anti-virus software systems use a signature deployment mechanism via the Internet which supplies signature updates to subscribers [2]. A similar model could be implemented for attack program signatures.

Figure 4.9 shows a distribution scheme. A central repository serves as storage for the signatures produced by signature-generation sites. This repository is mirrored by other Internet

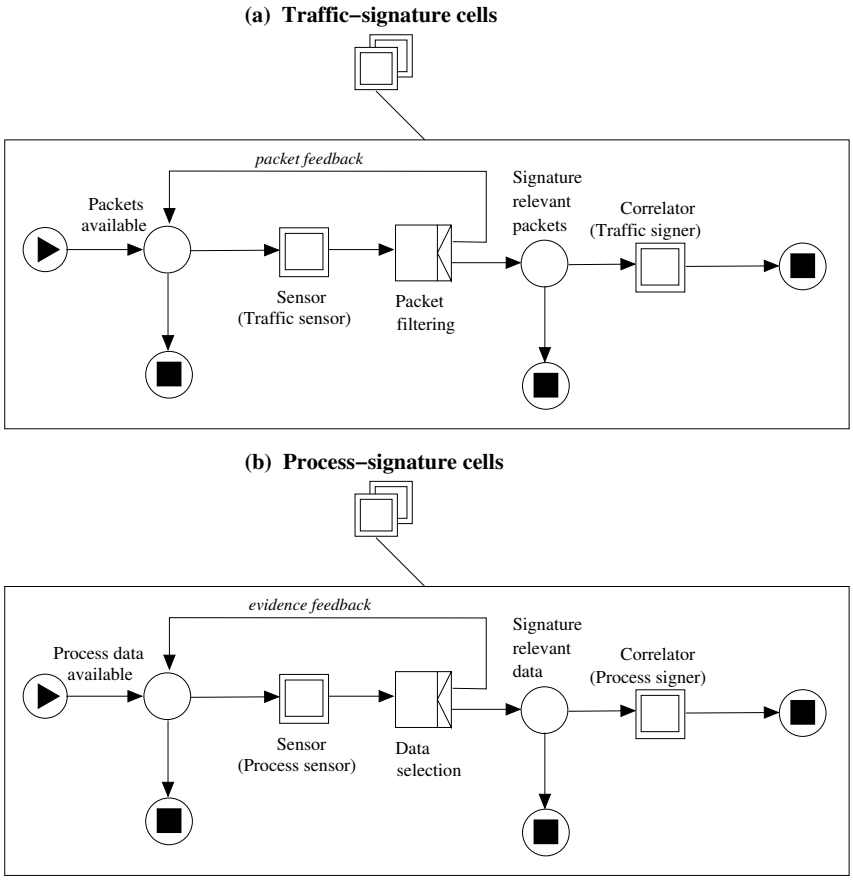


Figure 4.6: Internal dataflow: (a) Traffic-signature cells, and (b) Process-signature cells (see Appendix A for YAWL symbols).

servers which are accessed periodically by agents in order to obtain the most up-to-date version of the signatures or the ontology (an ontology for intrusion detection is rather static; there is no need to generate an administration agent that answers queries or automatically maintains the ontology [31]). Cryptographic certificates can be used to sign the bundles containing the signatures so that agents are able to verify their integrity and authenticity.

4.8.3 Operational outline

Summarizing, two data streams are the input to the architecture: outbound network traffic and process execution data (Figure 4.2). For signature generation, signature cells access these streams and extract information at the time the attack tool is executed. They share these signatures with local monitoring agents and alternatively download signature updates from an Internet signature repository (the repository is maintained by registered signature generation sites;see Figures 4.5 and 4.9). Traffic monitoring cells permanently screen network packets looking for a match with the existing signatures. In parallel, process monitoring cells do the same accessing the process data stream. Either of these two monitoring cells may identify an

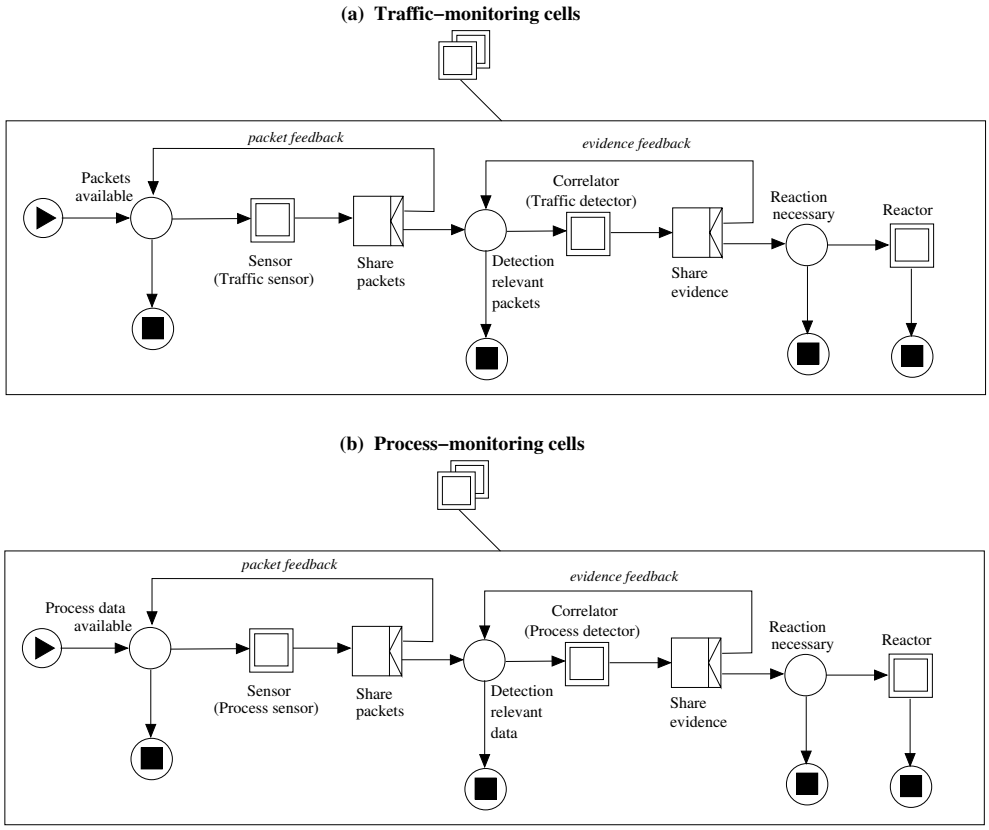


Figure 4.7: Internal dataflow: (a) Traffic-monitoring cells, and (b) Process-monitoring cells (see Appendix A for YAWL symbols).

attack tool and react accordingly. An additional correlation cell receives evidence from the monitoring cells, which it integrates in order to more precisely identify the attack program that has been detected.

4.9 Ontological model

The OID problem approached from a multiagent perspective presents the need for a common understanding of concepts and relationships, and for a more simple and meaningful integration with other security technologies such as firewalls and vulnerability scanners. Given the amount of data that needs to be analyzed, it is also necessary to filter and optimize communication contents in order to speed up all tasks. An ontology also helps reduce the size of the messages exchanged, limits redundancy, and allows for a more intelligent information analysis [204]. (When describing ontology components graphically, bold arrows will represent subclass relationship and soft arrows will represent attributes; see figure 4.10.)

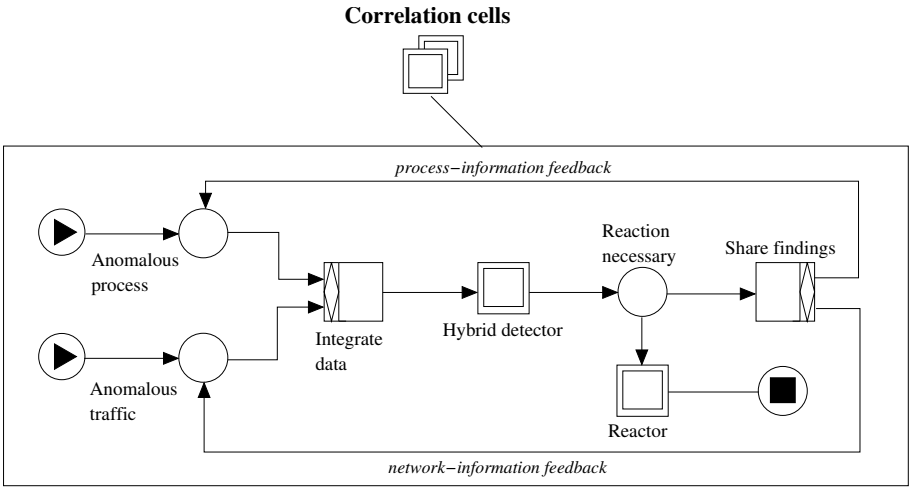


Figure 4.8: Dataflow inside Correlation cells (see Appendix A for YAWL symbols).

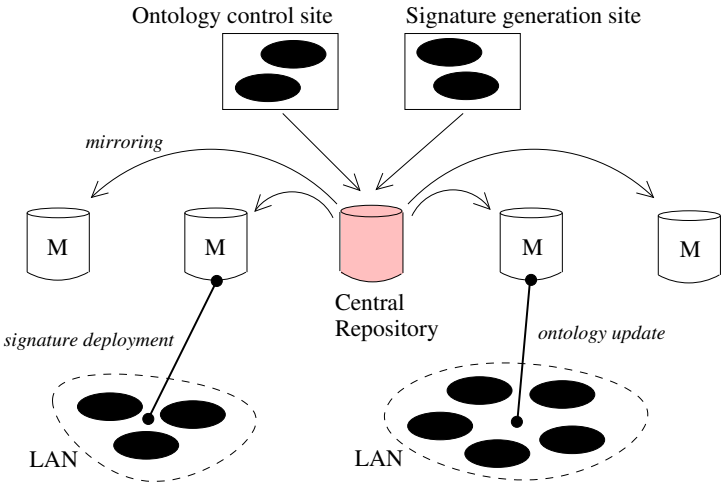


Figure 4.9: Deployment of attack program signatures. Repositories connected to the Internet receive signatures from registered signature generation sites. Agents download signatures and ontology updates (M represents a mirror repository).

4.9.1 Supporting agent-based security technologies with ontologies

Having a central specification of the structure of a data set helps automate the verification of the input for compliance with the structure. This is a basic benefit of using ontologies, however, its main advantage appears when logic processes need to be implemented and when data is being exchanged among different system components.

Most research in intrusion detection has been in taxonomic models. A taxonomy is always part of an ontology but, in addition to the domain classification benefits it offers, ontologies

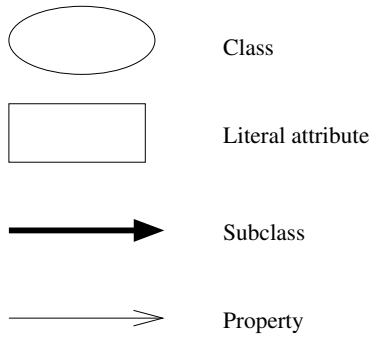


Figure 4.10: Graphical notation

provide constructs that allow a system, in this case an OID system, to reason over existing instances of the taxonomy [203]. They constitute a meta-data framework of the objects of the environment and describes them with all their features and relationships.

Relationships and attributes allow to infer information not explicitly specified when communicating a message. For instance, if the ontology establishes that (in RDF notation [203]): `(Agent, subclass, Reactor)` and also that `(Reactor, subclass, Tracer)`, an agent cell may correctly infer that a given `tracer` agent will inherit the functionality of `Agent` and `Reactor` as well even when this has not been explicitly said about `Tracer`. This information needs not be communicated, but only the basic facts.

An ontology for an intrusion detection architecture is necessary to enable more intelligent behavior in agents, to optimize communication contents and interpretation, and to give formalism to the way the components of an architecture interact (an aspect that is usually ignored). "... An ontology decouples the data model representing an intrusion from the logic of the intrusion detection system"[203]. From a software engineering perspective, this facilitates the integration with other technologies by sharing data in a standard format that can be equally conceptualized using the machine-interpretable constructs of the ontology.

4.9.2 Target-centric ontology for intrusion detection

In the area of intrusion detection there have not been very many efforts aimed at defining ontological schemes that enhance and simplify the interaction of operating systems and security applications [196, 43]. The most relevant project is the one by Undercoffer et al. [202, 203]. They defined a target-centric ontology for intrusion detection which is based on the analysis of over 4,000 intrusion instances. Figure 4.11 shows a representation of this ontology and Figure 4.12 shows the ontology for OID which is proposed in this dissertation and which is explained in the next sections.

The target-centric ontology project defines intrusion detection concepts from the point of view of the victim claiming that intrusion detection systems should exclusively handle information that can be measured and observed locally [202]. This same perspective is used in OID but, in this case, the position of the attacker is at a local host: the perspective of the attacking party in OID is equivalent to the perspective of a victim in intrusion detection models. Only locally observable information is used in OID to characterize a program's behavior and the network

traffic it generates.

The taxonomic structure of this ontology defines three main classes:

- *target of attack* = {*network, kernel space, application, other*}
- *means of attack* = {*input validation, exploits, configuration*}
- *consequences of attack* = {*DoS, user access, root access*}.

that are used to describe how an attacker compromises a target through a certain means and this produces some consequences. All the relations among these classes are based on the way the set of analyzed intrusion instances behaves and covers local-to-local as well as remote-to-local scenarios.

There is a point where this intrusion detection ontology and an OID ontology intersect. This point is the *Network* resource (Figure 4.13). From the intrusion detection perspective, *Network* is a subclass of *Location* and represents a potential point from which a host can be attacked. From the OID point of view, *Network* constitutes a *data source* that needs to be monitored by a sensor agent in order to collect relevant information (see Figure 4.12).

Subject–predicate–object structure

An ontology can be understood as a vocabulary that defines the terminology of concepts used by agents and the nomenclature of the relationships between them. It defines the semantics and structure of an integrated agent-based system and allows for inference processes to access known facts in order to create new knowledge from them.

The description of the OID ontology that appears next follows the $(Subject, Predicate, Object)$ RDF statement convention [203]. Let R be a set of system *Resources*, $P \subseteq R$ a subset representing resource *Properties*, and L a set of literal values (e.g., not subjects or classes, but simple values like integers or strings). Then $Subject \in R$, $Predicate \in P$, and $Object \in (R \cup L)$. This relationships can be also described graphically.

This ontology has been defined in the OWL ontology language [7]. OWL extends RDF with richer constructs and allows to specify the meaning of classes and properties in various ways, so that intelligent programs can reason about it. OWL has been chosen because is one of the most recently developed ontology languages, it has undergone a more rigorous review process than other languages such as DAML+OIL and even RDF, and because it "has received significant input to its design from researchers with experience in building inference engines" [7].

OWL documents are typically stored in terms of an XML/RDF file that contains: (1) *classes* to define concepts from the domain, (2) *individuals* to represent specific instances of the classes, and (3) *properties* define the values each individual can take.

For instance, the statement `(Signature, subclass, Process_Signature)` which defines a subclass predicate would be represented in OWL as shown in Figure 4.14. (The document preamble defines some name-spaces such as RDF, RDFS, and OWL itself used to correctly reference objects from those domains).

In the case of an individual of the *Traffic Signature* class, which stores the representation of attack-traffic signatures introduced in the next section, its attributes could be graphically described as shown in Figure 4.15.

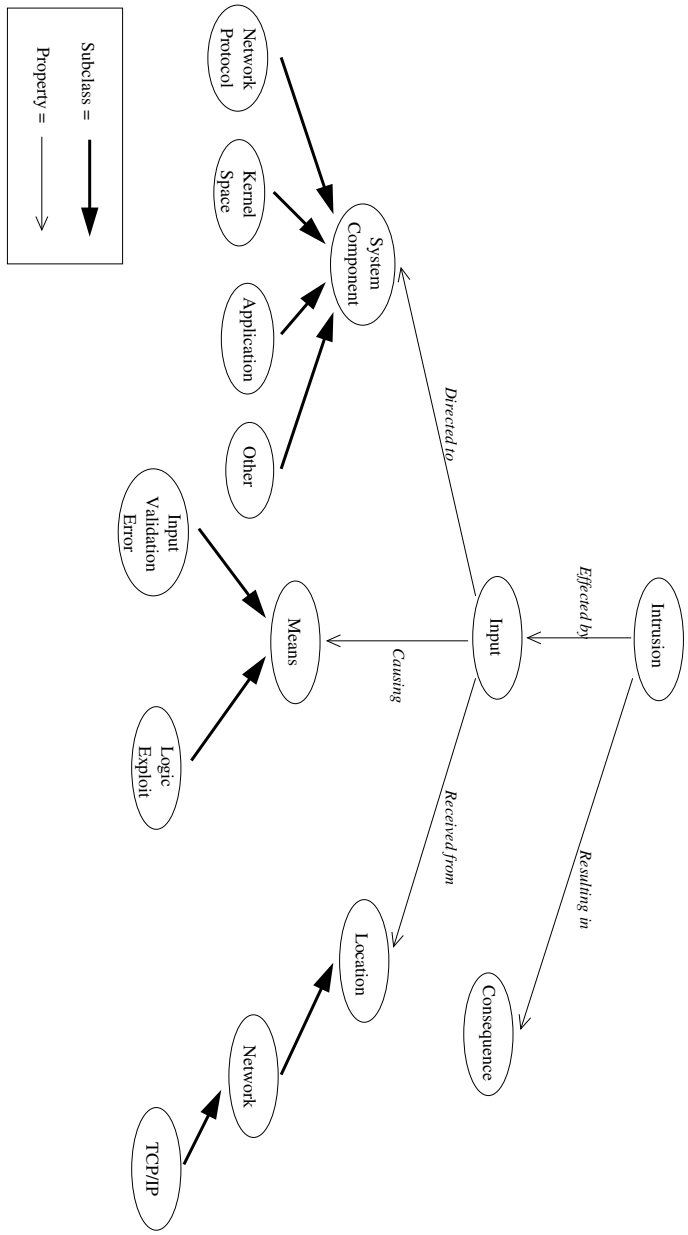


Figure 4.11: A target-centric ontology for intrusion detection [203, 202]; bold arrows represent subclasses; soft arrows represent properties.

Literal attributes qualify the subjects of a class. They represent atomic features that need no further organization or classification and that are directly attached to the subject in question (e.g., the size of a packet payload; it contains an integer number which does not have any

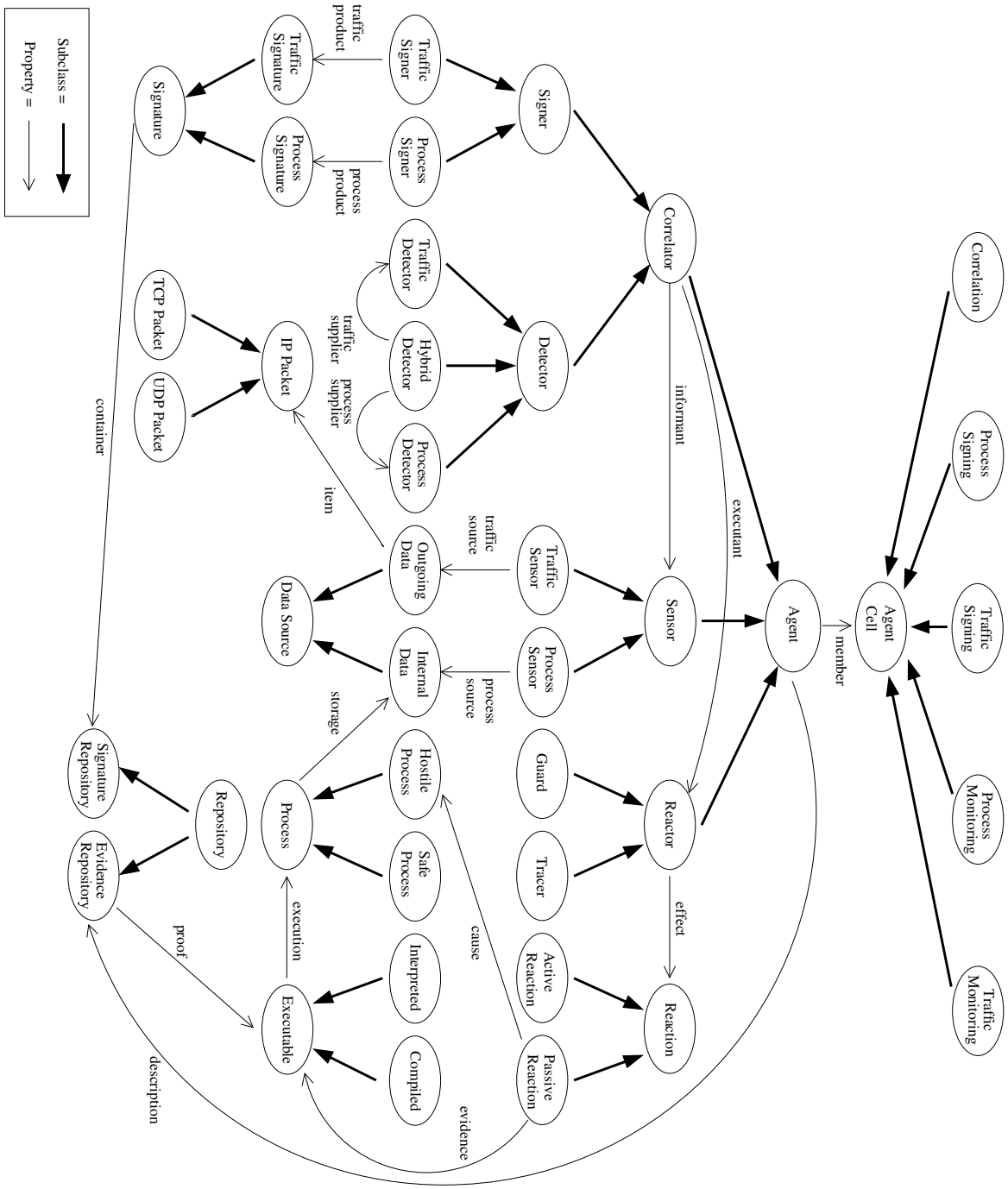


Figure 4.12: A source-centric ontology for outbound intrusion detection (see Appendix B for its definition in OWL)

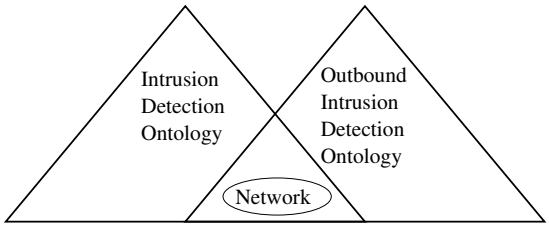


Figure 4.13: Overlapping between the target-centric intrusion detection ontology and the source-centric OID ontology.

```
<rdf:RDF
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    <owl:Ontology rdf:about="OID Ontology">
        <owl:imports rdf:resource="http://protege.stanford.edu/plugins/owl/protege"/>
    </owl:Ontology>

    <owl:Class rdf:ID="ProcessSignature">
        <rdfs:subClassOf>
            <owl:Class rdf:ID="Signature"/>
        </rdfs:subClassOf>
    </owl:Class>
...
</rdf:RDF>
```

Figure 4.14: OWL statements defining the Signature–Process Signature subclass predicate.

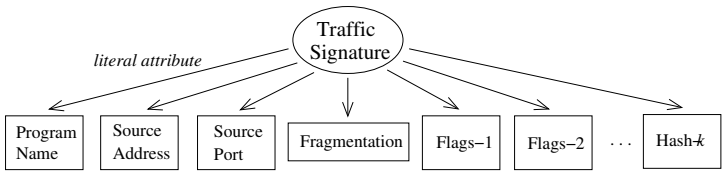


Figure 4.15: Literal attributes of the *Traffic Signature* object.

sub-categories or attributes itself).

If the parameters defining a traffic signature changed, the ontology would have to be updated and the agents would receive this structural modification through the signature deployment system. For instance, if we wished to include a new $programID$ attribute into *Traffic Signature*, the class would change and an individual, "tesoIISb" in this case, would look like in Figure 4.16.

These structures allow all architectural components to equally interpret what happens in the environment. The relationships between an exploit program and a file, a signature-generation agent and a data source, a signature and a repository, etc., can be all clearly defined. This

```
<TrafficSignature rdf:ID="tesoIISb">
   <programName>tesoIISb</programName>
   <fragmentation>1</fragmentation>
   <sourceAdddress>127.0.0.1</sourceAdddress>
   <sourcePort>0</sourcePort>
   <destinationAddress>132.243.46.41</destinationAddress>
   <destinationPort>80</destinationPort>
   <windowSize>5480</windowSize>
   <flags>0,8,1,10,1,10,1,1,1,1,1,0,0,0,0,0</flags>
   <hashes>0,0,0,0,1528,1820,0,4104,0,1739,0,0,0,0,0,0</hashes>
   <programID>50</programID>
</TrafficSignature>
```

Figure 4.16: OWL definition of a sample Traffic Signature instance.

reduces ambiguity, makes communication simpler, and creates an interface other agents and software applications can access for communicating with the OID architecture.

4.10 Source-centric ontology for outbound intrusion detection

The first part of this section describes the taxonomic structure of the proposed ontology including object classes, subclasses, and instances. Instances are shown only for subclasses with three or less instances given that some subclasses can potentially have hundreds of instances, like in the case of the IP packet class. The second part of this section outlines the relations among objects of the ontology. The whole contents of the ontology appear in Appendix C.

Taxonomic structure

The components of this ontology are based based on the commonalities found among the different intrusion detection architectures surveyed ([224, 41, 158, 92, 221]). These commonalities were extended with the particular points of the OID approach. The ontology contains nine primary classes that cover all the objects that appear on an OID scenario: *agent class, agent-cell class, data-source class, signature class, reaction class, process class, repository class, IP-packet class*, and *executable class*.

1. **Agent-cell class**. An agent-cell is composed of one or more agents to perform a certain task. There are five types of agent class: correlation cells, (2) process-signing cells, (3) traffic-signing cells, (4) process-monitoring cells ,and (5) traffic-monitoring cells.

2. *Correlation cells* develop integration and interpretation functions. They are composed of a sensor, a correlator, and a reactor agent.

3. *Process-monitoring cells* are responsible for observing the behavior of processes in order to extract data for the detection and signature generation cells. They are composed of a sensor, a correlator, and a reactor agent.

4. *Process-signature cells* receive process information in order to define process signatures. They are composed of a sensor and a correlator agent.

5. *Traffic-monitoring cells* are responsible for observing outbound traffic generated by suspicious programs and supply this data to the detection and signature generation cells. They are composed of a sensor, a correlator, and a reactor agent.

6. *Traffic-signature cells* receive network packets in order to define traffic signatures. They are composed of a sensor and a correlator agent.

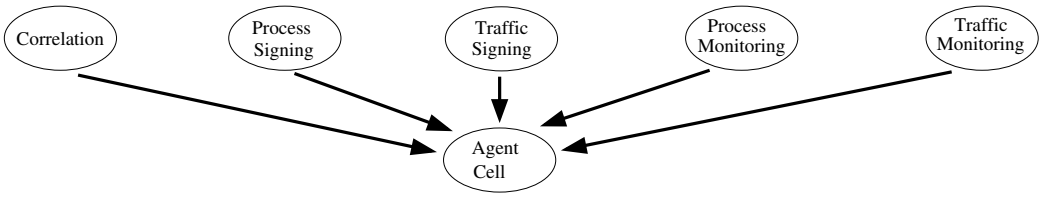


Figure 4.17: The Agent-cell class

7. **Agent class**. There are three main agent types: (1) sensors, (2) correlators, and (3) reactors. These, in turn, have two subclasses each (Figure 4.18).

   Sensors can be: (1) *process monitors*, or (2) *traffic monitors*. This is because a program will be modeled according to its execution as a process and also through the network packets it generates.

   Correlators can be: (1)*detectors*, or (2)*signature generators*. While signature generators receive input from sensors with the objective of defining program signatures, detectors receive input to detect the occurrence of such signatures.

   A *reactor* agent can be: a (1) *tracer*, or a (2) *guard*. Guards implement local reaction to incidents by updating the environment or by passively increasing the monitoring level over a certain resource. Tracers cover the function of back-tracing the connection of an intruder in case an interactive session has been established with a local host.

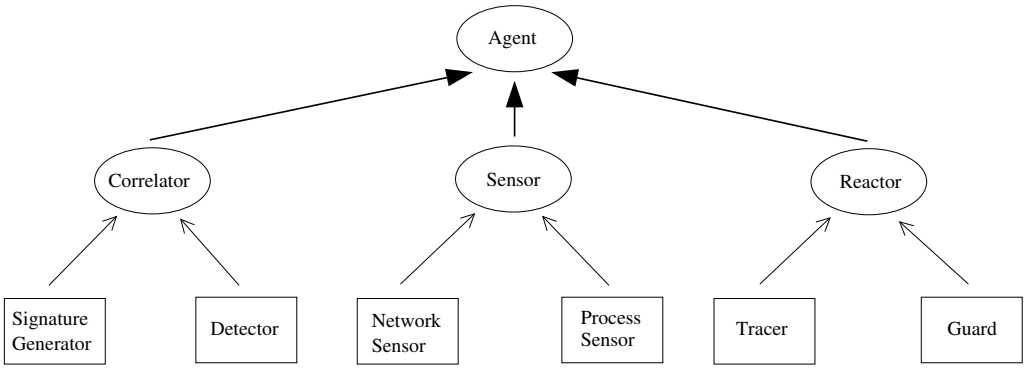


Figure 4.18: The Agent class

8. **Data-source class**. Data sources are classified into two subclasses: (1) data sources for external use, and (2) data sources for internal use (Figure 4.19). It is the type of the ultimate information consumer what determines the essence of the data from a security-monitoring perspective.

   Sources that produce *data for internal use* (e.g., the kernel process table) can be locally accessed. Although a remote party could also access a local system to submit a query, these sources are not intended for external use.

   There are sources that generate *data for external use* (e.g., outgoing network packets). This information has no internal meaning unless it is used for debugging purposes (incoming traffic is not being considered as it does not have the significance to OID outgoing traffic does). In the case of OID, this is a basic information resource.

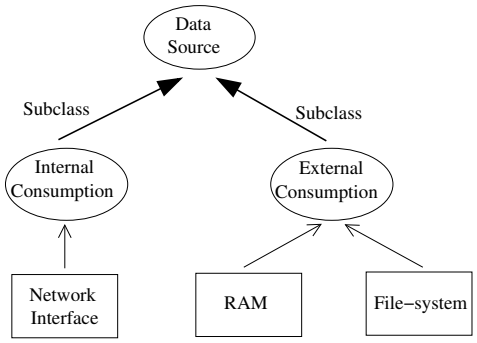


Figure 4.19: The Data-source class

9. **Signature class**. There are two signature types utilized within the OID architecture: (1) traffic signatures, and (2) process signatures (Figure 4.20). These two types of pattern can be used independently or jointly to detect an attack tool.

   A *process signature* contains the execution properties of a program, in particular, regarding the effects it has on local resources while in execution.

   A *traffic signature* characterizes the type of network traffic generated by an attack tool when it is executed. Notice that both signatures are used to characterize the same object (i.e., an attack program).

10. **Reaction class**. There are two types of reaction to incidents: (1) passive reactions, and (2) active reactions (Figure 4.21).

    Passive reactions do not take any steps toward stopping an attack attempt. They focus on collecting additional information that can be used as evidence in the future (back-tracing is an example of passive reaction).

    The goal of *active reactions* is to counter an intrusion attempt by a user who is executing an attack program locally toward an external victim. This class of reaction has more impact on the environment than the mere information-oriented goal of passive reaction.

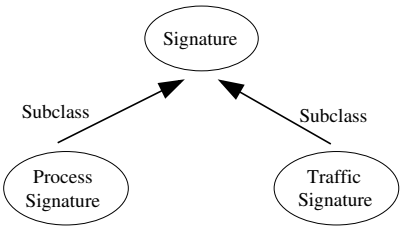


Figure 4.20: The Signature class

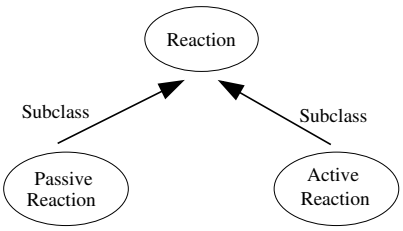


Figure 4.21: The Reaction class

11. **Process class**. A process is a key object for OID. It represents the execution of an attack program and contains critical information for the detection algorithms. There exist two different types of process in this ontology: (1) legitimate processes, and (2) hostile processes (Figure 4.22).

    This categorization is based on matches found by a detectors. *Hostile processes* are subject to further inspection, whereas *legitimate processes* do not have to be monitored once it has been verified they do not pose any threat to the security of other systems.

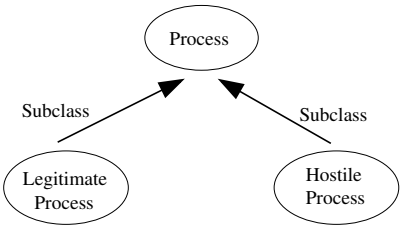


Figure 4.22: The Process class

12. **Repository class**. There are two different types of repository: (1) signature repositories, and (2) evidence repositories (Figure 4.23).

    The purpose of an *evidence repository* is to store information that helped correlators to reach intrusion-related conclusions.

    A *signature repository* is in charge of storing the program signatures generated by the signers. A repository is a passive entity, and as such, its administration is not autonomous. Agents

are in charge of keeping these repositories up to date (alternatively, a security analyst can be in charge of maintaining this type of repository).

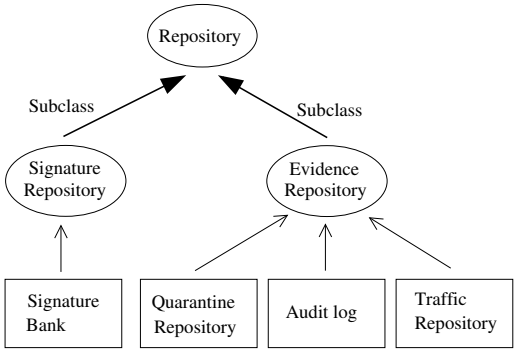


Figure 4.23: The Repository class

13. **IP Packet class**. Network packets are key to signature definition and detection. This ontology supports two subclasses of IP traffic: (1) TCP packets, and (2) UDP packets (Figure 4.24). IP traffic counts for a big percentage of the attacks performed over the Internet [143] and the ontology could be extended in case other protocols needed to be supported.

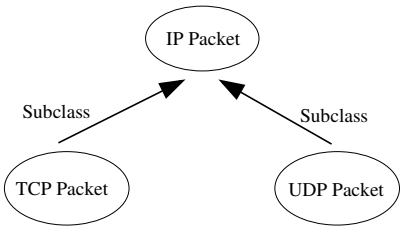


Figure 4.24: The IP-packet class

14. **Executable class**. The proposed OID approach is based on the execution of programs. There are two main categories we are considering: (1) attack programs which execute via an interpreter, and (2) attack programs that are compiled into binary format and can be understood by the host (Figure 4.25).

    It is necessary to split the Executable class into these two categories because the analysis that needs to be performed for each of them is different and because the information exchanged among agents is dependent on the type of executable that is being inspected.

4.10.1 Use of the ontology

The ontology is used by agents to process messages and to derive information not explicitly conveyed to the listener during a communication. There are two main tasks: (1) message interpretation and (2) consistency verification.

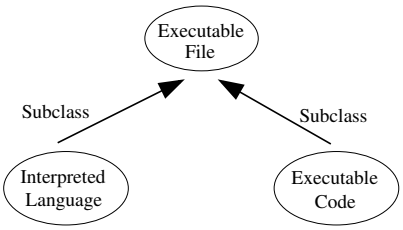


Figure 4.25: The Executable class

- **Message interpretation**. All messages exchanged between agents are based on the OID ontology. The sender builds the message using ontology classes in no particular order and filling out attribute values. The receiver expects to see certain message types depending of its own agent type and therefore looks for statements that are meaningful to itself. A major advantage of this approach is that the knowledge of the agent (i.e., the ontology and its knowledge base) is used to reduce the amount of data that needs to be exchanged.

  For instance, when preparing a message the sender simply specifies its agent ID and the listener will be able to fetch its agent type from its own knowledge base – updated every time a cell is spawned. From this single piece of data it is possible to infer parent classes and relationships the sender keeps with other system components.

  When a traffic signature is open by a *Traffic Detector*, the agent may require additional information regarding the corresponding process signature. Specifically, it will have to know the originating *Process Signer* that created the signature. Even when the *Signer ID* is not explicitly indicated on the message, the ontology indicates a traffic signature is generated by a certain *Traffic Signer* and the agent will have to first query its ontology and then its knowledge base in order to find the specific agent that will in turn point it at the right direction to find the process signature of the program.

- **Consistency verification**. When a message is received by an agent, the ontology is used to perform automatic consistency verification. Not only does it allow the receiver to check the integrity of the classes involved in the message (i.e., whether or not subject-to-property relationships are maintained), but also it can verify that the attribute values are valid and reflect the current state of the system.

  In its simplest way, the verification mechanism will check every message for compliance with the ontology by analyzing the structure of each of the classes included. For some classes this is a basic requirement as any inconsistency can and will affect the performance of some agents (e.g., the contents of a traffic signature; it must be complete and correct).

  Before going ahead with passive reaction, a *Reactor* may verify if the process that triggered the detection mechanism is still running. In order to do this, the *Reactor* will check the ontology and will know that a reaction must have been generated by an executable file. This executable may have a process attached to it and the agent will check whether the process is still alive in order to proceed with process suspension or simply with a change of execution privileges of the file.

4.11 Session tracing

The tracing component of the architecture aims at identifying the hops (i.e., IP addresses or host names) of a connection chain that ends at our host. Connection chains are used by intruders to hide the origin of their connection. From an OID perspective, identifying the components of a connection chain helps warn other hosts about potential security threats.

Our system is composed of a number of tracing agents that communicate with each other sharing information on user connections [3]. There is an agent running at each host h_i and the ideal situation is having an agent running on every host of the network. Every agent has client and server capabilities that enable it to request information from other agents and to supply information to them. An agent is not useful by itself. It needs to collect information from the others in order to identify the links of a connection chain. The agent at the target host will eventually collect information from a number of its peers and will be in a position to inform to the local administrator about the possible origin of the connection.

The perceptions of a tracing agent are received from two main sources: (1) the state of the system regarding user sessions and connections, and (2) messages from other agents requesting or providing information.

The environment an agent inhabits is highly dynamic and provides the communication channels necessary for an agent to stay in touch with its peers over the network – the environment can also represent a threat to the tracing system if malicious users gain access to it.

As a result of the changes perceived by an agent, it can perform a number of different actions: (a) send messages to an agent, (b) request messages from an agent, (c) broadcast messages to all tracing agents, (d) generate encryption keys, (e) encrypt messages, (f) decrypt messages, (g) identify connection loops, (h) retrieve system status information, and (h) save its knowledge to a safe location.

4.11.1 Internal design

The knowledge of an agent is stored on a repository known as **knowledge base** (KB). This KB includes tracing host information and user data. A tree structure stores the information on the hosts that are part of a connection chain. Information regarding user names, login times, and dates is stored on a separate structure. These two elements represent all the knowledge an agent has about the state of its environment. Should this knowledge be lost, the agent would lose track of previous events and would have to be updated by its peers.

The actions an agent takes depend on the information of the state of the system and its perceptions. Embodied into the agent are a set of rules that enable the agent to work autonomously deciding what to do. As new knowledge is fed into the KB, more rules from the rule base need to be analyzed before making any conclusions and proceeding upon them. The type of perceptions and actions do not change but the state of the environment does evolve hereby modifying the behavior of agents. The type of rules that make up the rule base are of the form:

[3] This system was implemented in C and has been successfully tested separately from the Java agents that implement OID [118]; the integration of the two modules is still pending (see Section 7.3 in the last chapter)

```
R1:
IF ((HOW-CONNECTED(User)=Console) AND (WHAT-DOING(User)=Ssh-to(Target)))
⇒ TRACING-DONE();


R2:
IF ((MESSAGE-TYPE(Message)=Tracing-request) AND (VALID-PUBLIC-KEY(P-key)=False))
⇒ DENY-CONNECTION() AND BROADCAST-TREE() AND ALERT();
```

In order to protect their information exchanges, all agents have an RSA key-pair that is used to negotiate symmetric session keys before encrypting their messages [179]. Encryption at this level guarantees that the information will not be visible at any point before reaching the top of the IP stack of the recipient party. This will deter sniffing attacks that would otherwise be possible with agent communication in plain text [93].

The tracing process starts when a connection needs to be monitored. Agent A at target host h_t receives local information from the system and checks for remote connections. If there is a connection from a remote machine h_r, A will contact agent B at host h_r in order to research the connection. A private channel will be opened using the asymmetric encryption keys of the agents. A symmetric session key will be agreed upon and all messages will be encrypted with it for that exchange. The same procedure will be followed if there are relevant connections from outside of the host of h_r coming into the server. A set of secure channels will be setup by the agents in order to share the information they have regarding connections. One hop at a time, the links composing a connection chain will be found.

4.11.2 Hop identification

The tracing information obtained by an agent is shared with the rest of the agents through broadcast communication. This guarantees that the KB's of the agents are consistent with the state of the environment and that, in the event of a security incident, the knowledge acquired by an agent will not be lost as it has been replicated by other agents.

If we have a set H of n hosts, and a user creates a connection chain C of m hops, the system will behave well – with a *O(n)* worst case scenario. Suppose $m \gg n$. If the number q $(0 < q < m)$ of connections that go from host h_x into host h_y – or vice versa – form a loop, the arrangement of the agents will find at most two relevant connections between the hosts hereby unwinding the loop (e.g., a connection from A into B and a connection from B into A involving the same two user ID's). This is due to the fact that an agent is capable of identifying repeated connections toward itself by storing information on the connected users. In the case of loops passing through a total of k hosts, the chain will be reduced to at most $k-1$ hops. This makes a worst case scenario of *O(n)* with $n = |H|$.

The unwinding method is based on previously observed activity. An agent first stores user name and host-name information on its KB. When a tracing process is started, an agent will engage in communication with several neighbor agents in order to collect data that allow them to draw conclusions. Each time agent A receives new information, it checks its KB for loops. This is done as follows.

All tracing requests have an ID number w. Suppose agent A initiates a tracing round. If

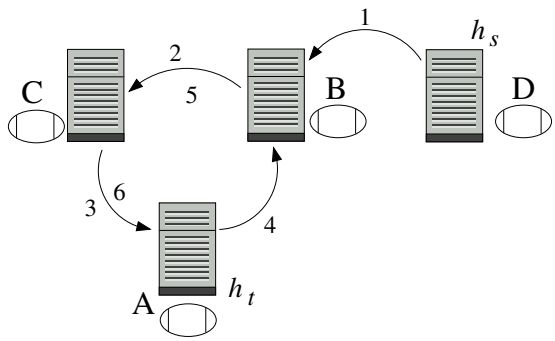


Figure 4.26: Loop unwinding. If a user connects from host h_s to host h_t doing a loop through hops 1, 2, 3, 4, 5 and 6 as shown above, agent A at h_t will start the tracing process and will be able to identify the loop to reduce tracing time in the future.

while waiting for news on a particular w the agent is being asked by a cooperating agent B for information to complete tracing round w as well, the agent knows it is part of a loop since it is receiving information on a tracing round that it itself started (all agents, including initiators have this property). The trace is stored on its KB tree T and, if the user being traced repeats a loop that passes through the same host where agent A is located, this agent will reduce the tracing time by omitting a further requests to follow a loop.

Once an agent F reports a local connection (e.g., connection at the console) when investigating a determined chain, the originating agent A concludes the search and reports the components of the $A \rightarrow F$ chain.

Chapter 5

Implementation

The First Resource for Outbound Intrusion Detection system (**FROID**) is a proof-of-concept implementation of the OID architecture proposed in previous chapters. It supports signature generation, detection, and correlation and has been developed using agent and semantic web technologies. This chapter discusses its design details and functioning. Experimental results are presented in Chapter 6.

5.1 Technologies

Multiple agent frameworks, knowledge representation languages, and automatic reasoning libraries exist. They cover specific functions that need to be combined with other tools in order to build a multiagent system capable of more intelligent behavior. The following is a brief description of the tools that were used in the implementation of this project.

- **JADE** (Java Agent DEvelopment framework) [17] is an open-source Java-based environment to build and host agent-based systems. It follows FIPA standards and contains libraries that provide different levels of communication and control functions that can be used to define the behavior of agents. JADE supports an ACL (Agent Communication Language) message exchange layer as well as a basic ontology that can be extended for specific applications.

  JADE lacks support for intelligent agent functions but, given that it is Java-based software, it can interact with other systems that do provide specialized functionality (e.g., Jena). JADE models can be distributed across several machines running different operating systems as support for agent location and mobility is also available.

  A GUI allows users to visualize agents, message exchanges, and the status of other system components. This simplifies debugging and allows for a more intuitive understanding of the agent platform.

- **Jena** is a framework for developing semantic Web applications [30]. Its API supports RDF-based languages such as DAML+OIL and OWL and contains a number of Java constructs for querying and manipulating data models. Search and navigation functions as well as set operations like union, intersection, and difference can be applied to the input knowledge base in order to locate objects and attributes or to answer other types of query.

Sophos Ex. 1007
Page 109 of 216

Its class methods for accessing RDF repositories make of Jena a valuable tool when integrating knowledge management capabilities into agents.

- **JPCAP** (Java Packet CAPture) is a network packet capture library [96]. The PCAP API that is available in C was ported to Java and became JPCAP providing almost the same functionality for packet capture and analysis. JPCAP supports several protocols through a set of Java classes that allow the user to read and modify any datagram. It also provides a graphical interface for doing traffic analysis in real time via `tcpdump` filters that taken as the basis for developing more elaborate traffic screening mechanisms.

- **Protégé** is an open-source ontology editor and knowledge acquisition system [144]. It allows the user do define an ontology, to instantiate classes, and to view knowledge bases through other applications. Protégé was written in Java and has multiple plug-ins for different ontology languages. It has been designed to guide developers and domain experts in the process of knowledge-based systems development through a set of modules that can be separately used.

  Protégé also allows the user to create, run, and save queries that return instances or object attributes from a knowledge base. It can verify instances through consistency-checking routines and identify immediate dependencies and transitivity between classes.

- **OWL** is a Web ontology language used to process information contained in documents rather than just presenting it to the user [7]. Like other ontology representation languages, OWL helps express the meaning of the objects that are part of an environment as well as represent their attributes and existing relationships. OWL provides more modeling primitives for expressing semantic contents and its design is based on lessons learned from the implementation and revision of the DAML+OIL ontology language.

5.2 Triggering event

The event that triggers the detection mechanism and the signature generation sequence is system call 117 on the Linux kernel 2.4.20-8: `opensock(int, int, char*)`. Whenever the kernel sees this type of request, monitoring cells are alerted and received the *PID* of the caller in order to begin inspection or attribute extraction.

`opensock` maintains the TCP socket table stored under the `/proc` pseudo-filesystem. A sample TCP socket table is shown in Figure 5.1. This table is permanently accessed by monitors in order to detect whether a new exchange is in progress – this includes both monitor types, network and process monitors. Each sensor agent reads up the packets it will review as well as the packets corresponding to the agents it is backing up (see description of the agent backup system below).

As mentioned in previous chapters, the FROID implementation focuses on TCP packets only as this protocol is widely used by attack and surveillance tools.

```
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid t inode
 0: 00000000:8000 00000000:0000 0A 00000000:00000000 00:00000000 00000000   29  0 1635 1 d2d8fa80 300 0 0 2 -1
 1: 0100007F:8001 00000000:0000 0A 00000000:00000000 00:00000000 00000000    0  0 2096 1 d362b080 300 0 0 2 -1
 2: 00000000:0203 00000000:0000 0A 00000000:00000000 00:00000000 00000000    0  0 2095 1 d362ba80 300 0 0 2 -1
 7: 8584110A:8108 E712020A:006E 06 00000000:00000000 03:00001639 00000000    0  0 0 2 c8c4e200
 8: 8584110A:8104 E612020A:006E 06 00000000:00000000 03:00000697 00000000    0  0 0 2 c8c4e280
 9: 8584110A:8106 EB06D894:0050 01 00000000:00000000 00:00000000 00000000  500  0 25559 1 c0c35a80 27 4 0 2 -1
10: 8584110A:8107 EB06D894:0050 01 00000000:00000000 00:00000000 00000000  500  0 25568 1 c0c35580 29 4 8 2 -1
    ...
```

Figure 5.1: Sample TCP socket table.

5.3 Agent creation

The core of the agents implemented for this project is based on JADE [17].

JADE provides agent name location and communication primitives that have been used to speed up the development of the system. Each agent cell is implemented as an JADE environment also called *container* which is in turn a Java virtual machine. Each environment executes agent-support classes that provide all basic methods for the functioning of the agent environment.

Each container executes multiple concurrent agents using threads and can communicate with other environments through message passing. An OID cell is equivalent to a single environment and contains a combination of agents as indicated in the previous chapter. Figure 5.2 shows the class inheritance structure utilized for deriving all agents:

A set of JADE *behaviours* define the functioning of each agent. *Behaviours* are routines used to group into modules the different methods an agent is able to execute depending on its state and perceptions. For instance, a $TrafficSensor$ agent contains a permanent-loop behaviour to monitor outgoing traffic as well as a timer behaviour that helps the agent determine whether it already has enough packets in the buffer in order to model a particular attack program:

```
public class TrafficSensor extends Sensor
{
   protected void setup()
   {
      addBehaviour( new LoopBehavior( this ) );
      addBehaviour( new TimerBehavior( this ) );
   }
...
```

Every agent obtains a copy of the ontology at start-up from a webserver known beforehand – no automation has been implemented to update the ontology given the static nature of the objects and relationships in the design of the OID environment; for our purpose, security is a priority over eficiency. The *model* variable stores the ontology and is used to interpret and compose messages at runtime:

```
   private final static String ontologyURL = "url:http://OIDhost/oidOntology.owl";

   public void loadOntology() {
```

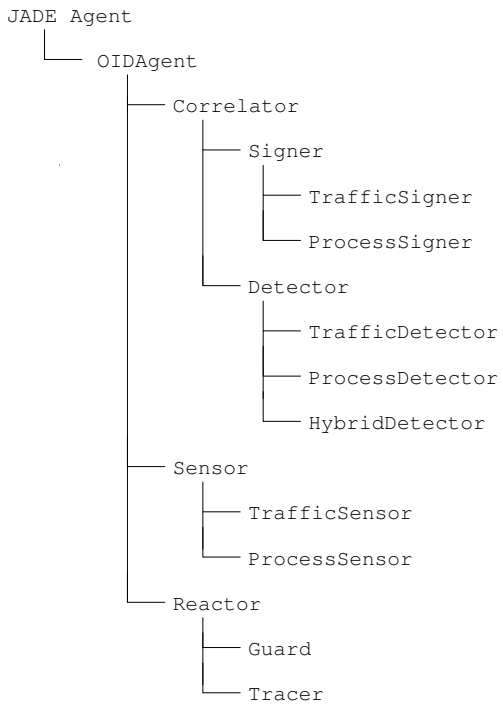


Figure 5.2: Hierarchy of classes derived from the primary JADE-agent class.

```
      model = ModelFactory.createOntologyModel( OntModelSpec.OWL_MEM, null );
      model.read( ontologyURL );
   }
...
```

5.4 Network data capture

Network sensors read outgoing network packets from the network interface via the JPCAP library [96]. There is no need for them to operate in promiscuous mode as only outgoing packets are being captured and they are available at the network device without the need of further privileges. For the FROID implementation all sensor agents use a `tcpdump` filter that screens TCP outgoing network packets:

```
filter = "proto TCP and dst not " + localAddress.getHostAddress();
```

In the above example, a variable stores a filter string in `tcpdump`-syntax and packets corresponding to the TCP protocol whose destination address is not the local IP address will be captured. Preliminary screening like this performed by network sensors is intended for correlation and reaction agents to receive only the packets that will mean something to them without having to spend time doing basic filtering by themselves.

During signature generation a sensor will capture up to 16 outgoing TCP packets. These packets come from an application that has requested a socket to send data to the outside. When an application generates less than 16 packets, the `packet counter` variable is updated as to

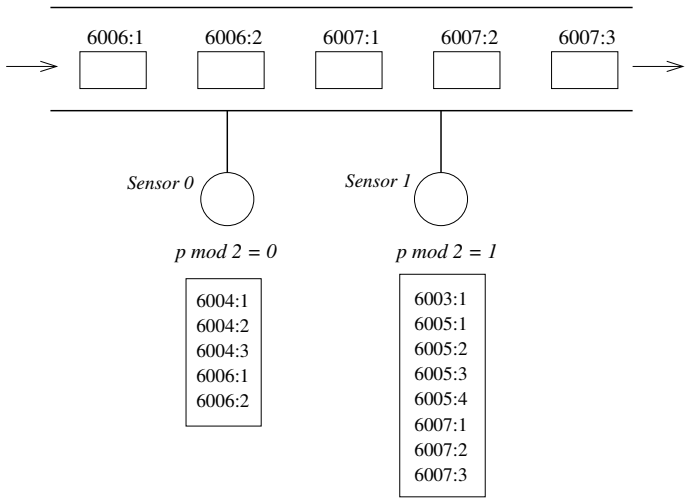


Figure 5.3: Packet reading assignment for two ($n = 2$) monitoring cells.

show this peculiarity. We have considered 16 packets in the design after having experimented with the execution of multiple attack tools. Most of them would send more than five but less than 17 packets toward their targets at launch time. In order to consider as many packets as possible without compromising performance by having to buffer a large number of packets once a socket is open, we use up to 16 packets and use a packet counter to describe how many packets the application generates.

Sensors capture packets and immediately forward them to correlation agents. Many concurrent sensors may be running and reading packets at the same time. When multiple monitoring cells are activated, monitoring is coordinated through a read backup mechanism described next.

5.4.1 Buffering and coordination

When a single sensor is in operation it may be the case that it misses some packets due to software failure or increased workload. In order to avoid this, the FROID system supports the execution of multiple, parallel monitoring cells.

Let n be the number of sensors available for network traffic monitoring. The packets each sensor sees are determined by the source port number p assigned to the socket being used. This is a consecutive integer number that increases as new sockets are requested [38]. In the case of $n = 2$, *Sensor* 0 and *Sensor* 1 will be in charge of reading up all outgoing TCP packets as a team. Modulo arithmetic is used to coordinate agents as follows:

$$p \; mod \; n \; = \; i \qquad \ldots (1)$$

where i indicates *Sensor* i will be in charge of reading the packet stream corresponding to source port p when n sensors are in operation. In Figure 5.3, *Sensor* 0 gets all packets corresponding to even port numbers whereas *Sensor* 1 is in charge of odd port numbers. As new sensors are added, n is updated accordingly so that there is always a primary agent responsible for a given socket.

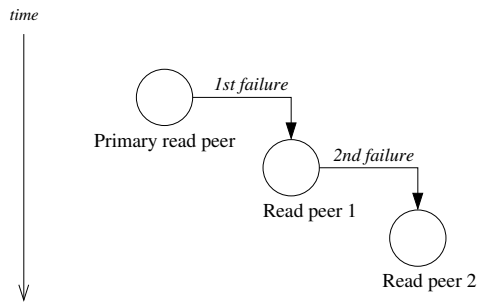


Figure 5.4: Failure cascade for sensors performing packet capture.

This decentralized structure of cells requires additional communication resources for agent location and message passing including a copy of the agent name table which, as the ontology definition, needs to be stored on every cell. In order to become operational as soon as possible, new cells are spawned from an existing cell of the same type from which they will inherit basic runtime data.

The addition of a new monitoring cells is communicated through broadcast. Although this represents an expensive communication approach to large-scale systems, the avoidance of central repositories it provides allows for increased robustness at the cost of more communication exchanges. Given the number of cells the FROID architecture is designed to support, this increased message exchanging is small (i.e., $m - 1$ where m is the number of active cells).

5.4.2 Packet read backup

A cascade mechanism allows a sensor to miss packets by designating backup agents that will support it in case of failure (Figure 5.4). The same mechanism is used for process sensors that, rather than handling network packets, extract data from processes. We now take the network sensor example to now explain the method.

Once a cell assumes responsibility for analyzing a certain socket number, it lets its read peers know so. A *read peer* is a backup monitoring cell that provides support to another similar cell in case it fails to read or scrutinize one or more packets. Each sensor is assigned two read peers making a total of three agents that will aid in the filtering of a single network transfer (if only one monitoring cell is active, no backup exists; in the case of two cells, they will function as backup of each other). Thus every outbound packet is always temporarily buffered by three different agents: the primary agent and its two read peers.

It is important to remember two design considerations made for this project: (1) the architecture does not prevent any outbound packet from passing through even when it may contain signs of attack; and (2) all buffered packets are discarded once a correlator receives them for detection.

Fault-tolerance principles suggest the use of three redundant entities to support stochastic failure [210]. Even when our entities are not redundant in the sense that they are not mirror copies of the primary agent but instead fully operational agents that provide data capture services to others, the principle still holds. For the sake of simplicity we have assumed probabilistically-independent failures in our implementation, although there may actually be

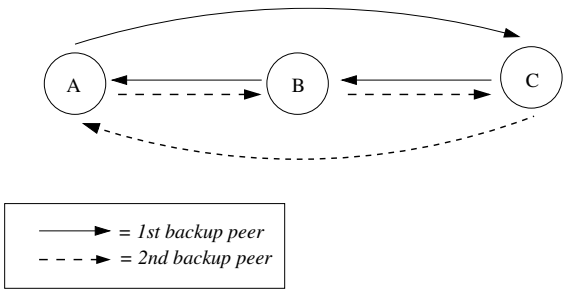


Figure 5.5: Sample backup structure for $m = 3$: when A is a primary agent, B is its first backup and C is its second backup; when B is a primary agent, C is its first backup and A is its second backup; when C is a primary agent, A is its first backup and B its its second backup.

some correlation between them (see Section 7.3 in the last chapter).

In case an agent fails to notify its intention to evaluate a transfer, its peers will not receive any message indicating the socket is already being taken care of. If a read peer receives such packet and has not received confirmation from the primary agent after two seconds, it will proceed to do the filtering and its cell will do the matching as it owns a full copy of the packets (Figure 5.5). The agent will announce this to the other read peer and, if it is itself processing a packet sequence, it will first finish with it before proceeding with its backup work. The second read peer is activated when it has received a packet corresponding to a primary agent but does not receives confirmation from the first neither the second agent after four seconds (two seconds for the primary agent failing to report, and two seconds for the second peer failing to report).

Only one pending transfer is allowed per backup peer (the agent works as first or second read peer only; cannot be both at the same time). A peer supports a primary agent when it fails but it is unable to support a second failing agent when it is already performing any backup function. At any point, an agent can only be assigned a maximum of three roles: primary agent, first read backup peer, and second read backup peer.

Similarly, once the primary agent is done doing signature matching with a stream of packets, it must announce this to its two read peers. The reason behind this consideration is the case in which an agent experiences trouble after it announced it will be processing a certain transfer but cannot complete the task. In that situation the same cascade procedure applies. The first backup peer will take over the operation and the second backup peer will do so in case the other two fail to report the matching. When an agent cannot report due to communication and not operational problems, it will complete its task but will not let the other agents know about it. In this case a packet stream will be checked more than once but that is more desirable than missing to do the checking.

5.5 Process data capture

Process data is derived from two main sources: (1) executable files and (2) the `/proc` pseudo-filesystem. Once a program has been identified as the originator of an outgoing packet sequence, the executable is used to extract the attribute values that will go into the process signature.

This information is first extracted from the executable file itself and confirmed with the data stored on the `/proc` file system.

The MD5 message digest of the text segment of the file needs to be computed in the first place. The text segment is extracted with `size`, then converted into a `String` and then passed to an instance of the MD5 class (no key is used with MD5 to create a MAC [179]; signatures must be verifiable by anyone):

```
import org.w3c.tools.crypt.*;
...
     Md5 txtSegmentMd5 = new Md5(txtSegment);
     String digestMd5 = txtSegmentMd5.getStringDigest();
```

The size of the text and data segments are obtained through `size` and then verified under the `/proc` directory corresponding to the `PID` of the process. If there is a discrepancy between the two values, the one stored under `/proc` is taken first as it reflects what the system is seeing. A discrepancy between the two reduces the weight of the attribute when matching is applied (see the signature matching section ahead).

The possible arguments to the program are then stored on the signature including the minimum and maximum number of arguments that can appear in the command line when the program is invoked. At detection time, this information is extracted from `/proc` as well.

The size of the executable, a strong attribute when other attributes match but a weak attribute by itself, is also recorded using the actual file of the attack tool which is located by the *Signature generation* and the *Monitoring* agents using the execution directory and file name derived from the PID.

5.6 Matching of network signatures

An attack tool can be executed using different arguments in the command line. Each combination of arguments makes the program behave differently by following a certain path inside the text segment and by varying the number and contents of the network packets sent to the target.

We took a test set of attack tools and executed them using different combination of parameters. Each combination or *variant* was executed three times so that process and network signatures could be generated for each variant. The objective was to evaluate the similarity of signatures corresponding to the same attack tool when using different parameters, and also to evaluate the similarity of signatures corresponding to different attack tools.

In the case of network traffic, we executed the same variant three times. For instance, we took attack tool A_{10} and compared the signatures produced by every execution. They were consistent except for hash values.

Hash values are computed by adding up the payload bytes of a packet. These values are checksums of the payload whose length is limited with $mod\ 10E6$ in order to preserve a fixed checksum size of six decimal digits, thus we refer to them as hash values.

The payload of the packet is not consistent from execution to execution as TCP wraps other packet types that contain variable fields. Such is the case of `telnet`, where the size of the payload is the same but its contents vary due to session negotiation flags. For that reason, a *deviation* field to represent the variation of the hash value due to sub-protocol flags is used. In

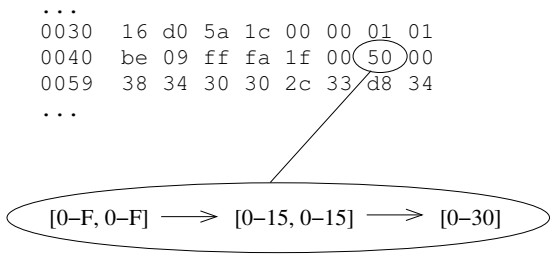


Figure 5.6: Payload data of a `telnet` packet. One byte is used for sub-protocol window size negotiation. The maximum value corresponds to FF which produces a deviation between 0 and 30 in the hash value.

the case of `telnet`, for instance, the deviation is an integer $d_{telnet} \in [0, 30]$. Figure 5.6 depicts this variation.

Network signatures corresponding to executions of the same variant are very similar. The *deviation* field helps capture the possible variations that may arise when transporting subprotocols.

We compared signatures corresponding to one single variant of different attack programs. Figure 5.7 is a representation of the signatures of three different attack programs using three attributes. Executions of the same program remain close to each other as explained before. The signatures corresponding to different programs are easily distinguishable as they form clusters that reflect the fact that the programs behave differently.

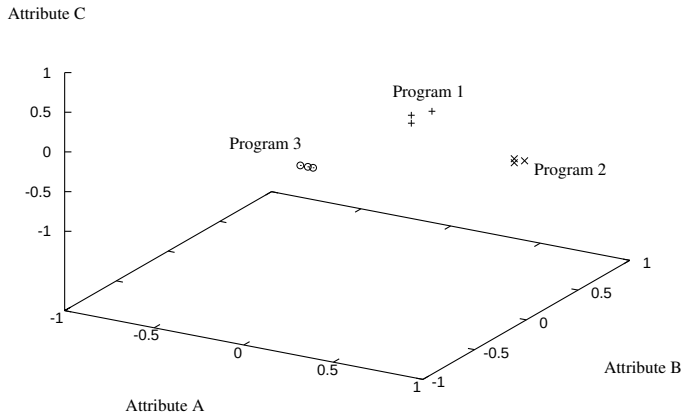


Figure 5.7: Network-traffic signatures of three different attack programs considering executions of *only one variant* per program.

When different variants for each program are exercised, the situation is different. They are not easily distinguishable for the same program and not even for the same variant. In Figure 5.8, for instance, a variant of *Program* 2 appears separate from the rest. However, the signatures of a second variant of *Program* 2 appear close to the cluster of a variant of *Program* 3. Even in a 3D representation, it is not easy to determine a single cluster that groups all the different variants of the same program. If there are over 100 programs and more than one variant per program, the situation is more complicated.

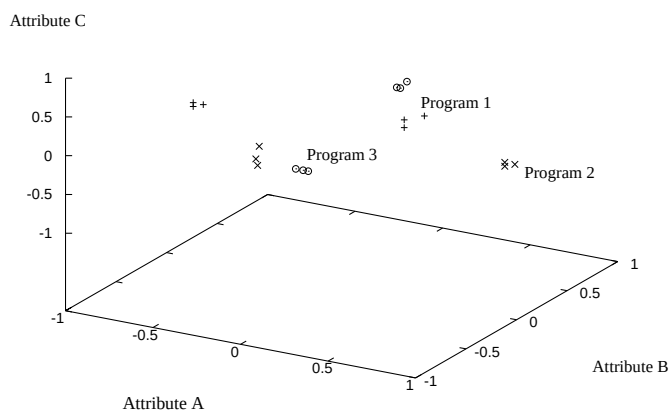


Figure 5.8: Network-traffic signatures of three different attack programs considering *two variants***.**

Consequently, our design contemplates program-variant signatures in order to capture the specifics of the of network traffic a program generates when using a different set of arguments rather than creating a single signature to group all variants together.

5.6.1 Primary data structure

Our network-signature matching algorithm is based on the work of Roesch [170] which was enhanced by Coit et al. in [37] using trees. We have modified the basic algorithm by considering the amount of information each field conveys. The idea is to reduce the time an agent spends matching its perceptions against its knowledge base in order to speed up response time.

The algorithm used by Snort groups attributes into two main node categories: (1)high-gain nodes and (2)low-gain nodes – the categorization is based on the experience of the authors in monitoring network activity. Our approach is to develop this categorization using experimental results from entropy and splitting power analysis applied to attack tool traffic so to use a data

structure that takes into consideration the different weights of its attributes and the frequency of its values – since read requests are much more frequent than read requests in this type of pattern-matching systems, it is possible to support a permanently-sorted data structure.

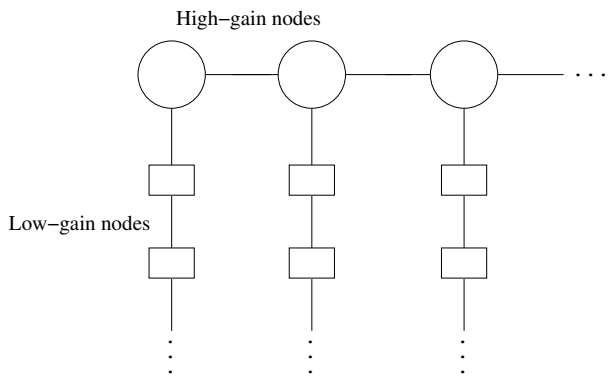


Figure 5.9: Snort's signature matching linked list [170, 37].

Figure 5.9 shows the bi-dimensional data structure used to store signatures in Snort. High-gain nodes contain a small group of heavy attributes. They are checked first for a match and, if there is one, the matching proceeds down the linked list where low-gain nodes are. These nodes contain the rest of the attributes that eventually lead to a classification of the input and each of them represents a signature. Once a new attack pattern needs to be added, new low-gain nodes are added and high-gain nodes updated.

Given the number of attributes we have, the data organization we used in the FROID implementation considers a third node type. In Figure 5.10 different node sizes reflect the fact that some nodes are more frequent than others. Node α_1, for instance, contains values that are common to a larger number of signatures than the values stored by node $alpha_3$. Nodes are placed in the linked list according to its weight, ω_i. Even though all α nodes contain the same attributes, the values they store give more relevance to some of them (i.e., if a certain combination of TCP flags helps easily classify packets, its splitting power will likely be high and it will be part of an *alpha* node at the front of the list).

β nodes are also sorted by frequency but their attributes contain attributes with less splitting power than α-node attributes. Unlike Snort, β nodes does not represent individual signatures. They group attribute values common to several signatures. If a match with an α node and a match with a β node have been found, the algorithm now tries to match γ nodes with the input. γ nodes do represent actual signatures and they are added to the corresponding β node list once a new signature needs to be considered. Important to recall that this method classifies packet sequences and not individual packets as Snort do.

In order to determine the order of attributes to build our three-dimensional linked list we have analyzed data obtained from attack tools.

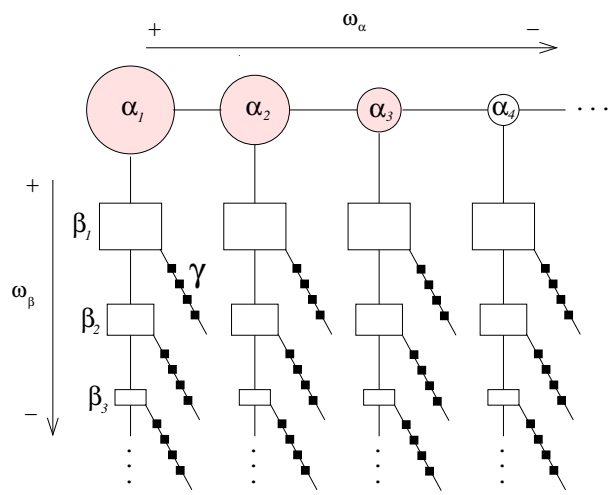


Figure 5.10: Data structure for attack signatures. Nodes are sorted from highest to lowest frequency.

5.6.2 Splitting power analysis

There are three splitting functions commonly used in classification trees [130]. These functions determine the amount of information a certain attribute conveys so that the most compact tree can be created.

Regardless of whether the tree creation technique is a greedy or multi-pass refinement one, it can be used to extract information about the weight of attributes for classification purposes. The three functions we used to compute the relevance of each attribute classification-wise are:

- Entropy. Represents the expected number of bits needed to encode the class of a randomly drawn sample using information theory coding:

$$Entropy(S_i) = \sum_{v \in Values(S_i)}^{k} -p(v)\ log_k(p(v)) \qquad \ldots (2)$$

- Information Gain. Computes the reduction in entropy due to the splitting on an attribute).

$$InfoGain(S_i) = Entropy(S_i) - \sum_{v \in Values(S_i)} \frac{\mid S_{i,v} \mid}{\mid S_i \mid} Entropy(S_{i,v}) \qquad \ldots (3)$$

- Gini index. Measures the impurity of a node considering how many samples go though each of its branches.

$$GINI(S_i) = 1 - \sum_{v \in Values(S_i)} p(v)^2 \qquad \ldots (4)$$

where S_i represents attribute i; $v \in Values(S_i)$ is a class for attribute i (continuous values are discretized in order to define a finite number of attribute classes); $|S_i|$ is the cardinality of the attribute defined by the number of input samples it receives; $|S_{i,v}|$ is the cardinality of each attribute class defined by the number of input samples that go into that class; and $p(v)$ is the probability of a random input sample being classified in class v as per the evaluation of its i-th attribute.

Let σ_j represent a splitting function. Different σ_j functions generate different trees, however, they all identify the most relevant attributes so that they can be placed at a shallow depth in the tree hereby helping more rapidly predict the classification of an input sample and reducing, at the same time, the tree's depth.

The use of more than one splitting function allows to reduce the selection bias of each technique and to identify high-value attributes with increased accuracy [130]. In order to integrate the three entropy values we use the following expression with domain in $\mathbb{R}$ and codomain in the $[0, 1]$ closed interval:

$$\lambda_{\sigma_j}(S_i) = \frac{\mid \sigma_j(S_i) \mid}{Max_{\sigma_j} - Min_{\sigma_j}} \qquad \ldots (5)$$

where S_i represents the i-th attribute, σ_j the j-th splitting function, $j \in \{1, 2, 3, ..., k\}$, k is the number of splitting functions being used, and Max_j and Min_j are the maximum and minimum values generated by the i-th function, respectively. Notice that the functions we use are either asymptotic or have an absolute maximum. Their values can be normalized to 1 so that they can be integrated into a final weight value Λ:

$$A(S_i) = (\sum_{j=1}^{k} \sigma_j(S_i))/k \qquad \ldots (6)$$

5.6.3 Analysis of network data

The decision trees produced with each function appear in Appendix D. They were generated using the decision tree module of the Gnome Data Miner tool [69] which implements the selected splitting functions. Table 5.4 shows the splitting-power values obtained for each of the attributes (Figure 5.11 plots the averages computed from the three splitting functions).

α ($A > 0.8$)	β ($A > 0.5$)	γ ($A \leq 0.5$)
fragmentation	destinationPort1	l7
windowSize	f5	l11
packetCounter	f6	h9
l4	sourceAddress	f16
h4	f8	f13
h5	sourcePort	h16
h6	h7	f12
deviation	h8	f15

l5	destinationAddress	f14
l6	f4	h12
f7	destinationPort2	l12
	f9	h12
	f10	l15
	f3	l10
	h14	f2
	h11	h10
	l9	l16
	l8	h2
	l14	h13
	f11	h3
		l2
		l16
		l3
		l13
		h15
		f14

Table 5.2: Attributes stored on each node type.

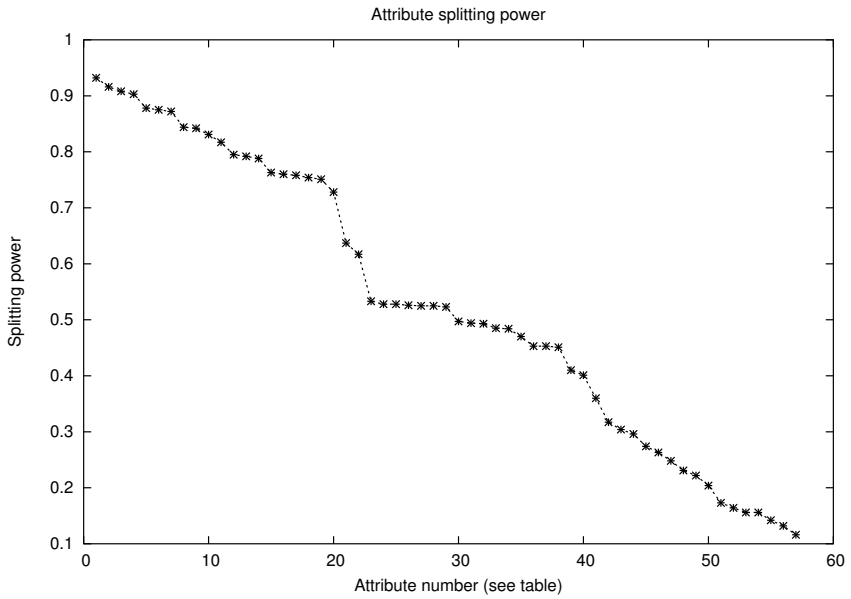


Figure 5.11: Splitting power per signature attribute.

	Attribute	*Entropy*	*InfoGain*	*GINI*	*Average (A)*
1	fragmentation	0.972	0.926	0.899	0.932
2	windowSize	0.902	0.922	0.923	0.916
3	packetCounter	0.874	0.886	0.965	0.908
4	l4	0.922	0.934	0.852	0.903
5	h4	0.920	0.822	0.893	0.878
6	h5	0.846	0.920	0.860	0.875
7	h6	0.824	0.924	0.869	0.872
8	deviation	0.869	0.899	0.764	0.844
9	l5	0.868	0.892	0.765	0.842
10	l6	0.837	0.829	0.826	0.831
11	f7	0.712	0.823	0.917	0.817
12	dport1	0.867	0.723	0.794	0.795
13	f5	0.768	0.783	0.825	0.792
14	f6	0.790	0.823	0.751	0.788
15	sAddress	0.817	0.725	0.748	0.763
16	f8	0.865	0.691	0.723	0.760
17	sPort	0.786	0.823	0.666	0.758
18	h7	0.743	0.725	0.794	0.754
19	h8	0.734	0.726	0.792	0.751
20	dAddress	0.753	0.689	0.743	0.728
21	f4	0.530	0.699	0.681	0.637
22	dport2	0.526	0.491	0.834	0.617
23	f9	0.634	0.509	0.455	0.533
24	f10	0.578	0.509	0.497	0.528
25	f3	0.578	0.483	0.523	0.528
26	h14	0.426	0.529	0.623	0.526
27	h11	0.590	0.560	0.425	0.525
28	l9	0.574	0.523	0.478	0.525
29	l8	0.593	0.447	0.530	0.523
30	l14	0.547	0.519	0.425	0.497
31	f11	0.535	0.425	0.523	0.494
32	l7	0.498	0.490	0.491	0.493
33	l11	0.425	0.500	0.530	0.485
34	h9	0.547	0.405	0.501	0.484
35	f16	0.464	0.426	0.521	0.470
36	f13	0.463	0.397	0.498	0.453
37	h16	0.434	0.225	0.699	0.453
38	f12	0.408	0.550	0.396	0.451
39	f15	0.400	0.356	0.474	0.410
40	f14	0.446	0.534	0.223	0.401
41	h12	0.385	0.285	0.409	0.360
42	l12	0.427	0.252	0.271	0.317
43	h1	0.366	0.489	0.057	0.304
44	l15	0.377	0.303	0.208	0.296

45	l10	0.328	0.125	0.370	0.274
46	f2	0.355	0.216	0.217	0.263
47	h10	0.019	0.037	0.688	0.248
48	l16	0.367	0.217	0.110	0.231
49	h2	0.185	0.293	0.188	0.222
50	h13	0.207	0.058	0.346	0.204
51	h3	0.186	0.205	0.129	0.173
52	l2	0.102	0.197	0.193	0.164
53	l1	0.157	0.190	0.123	0.156
54	l3	0.144	0.194	0.131	0.156
55	l13	0.128	0.144	0.155	0.142
56	h15	0.107	0.166	0.124	0.132
57	f1	0.132	0.108	0.108	0.116

Table 5.4: Splitting power per attribute in network signatures.

According to the results, we have selected the attributes with splitting power 0.8 or above to go into α_i nodes. Attributes whose splitting power is 0.5 or greater are stored on β nodes, and attributes with splitting power less than 0.4 are stored on γ nodes. Table 5.2 condenses these selection criteria.

5.6.4 Matching method for network signatures

The matching method is a score-based system. An overall score *sigma* represents the similarity between the input packets and a determined signature. This score is composed of a linear combination of sub-scores $sigma_i$ that correspond to a match between the i-th attribute of the signature and the input.

Each attribute contributes differently to σ scores (Figure 5.12). The weight ω_i of an attribute is determined by its splitting power which is also indicative of its relevance as a similarity measure:

$$\omega_i = ln(k) - ln(i) \qquad \ldots (7)$$

where k is the total number of attributes. Weights are combined into σ and then normalized to generate an output in the range $[0, 1]$:

$$\sigma \;=\; \sum_{i=1}^{k} \frac{\sigma_i \omega_i}{\sum_{j=1}^{k} \omega_i} \qquad \ldots (8)$$

All σ_i sub-scores return a value in the $[0, 1]$ range as well. In fact, all attributes return a boolean value, $\sigma_i \in 0, 1$ except for the *hash* attributes which are not checked for identical match as the others. As mentioned earlier, TCP sub-protocols may change the payload of a sequence and that is why the *deviation* attribute is used.

We use a triangular function to evaluate similarity of hash values. The maximum of the function corresponds to a match of hash values with zero deviation (i.e., perfect match; Figure

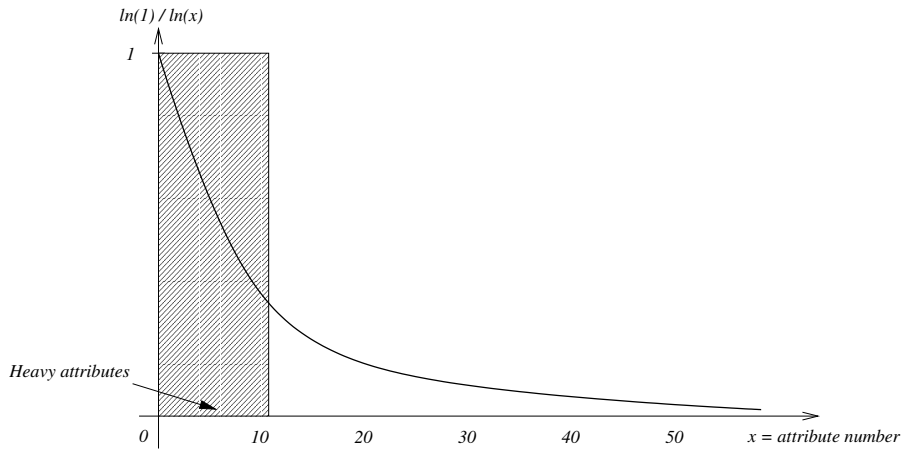


Figure 5.12: The matching method gives more weight to attributes with high splitting power (see Table 5.4).

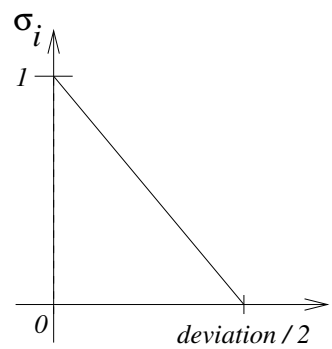


Figure 5.13: Function to compute payload hash σ_i sub-scores.

5.13). Any difference between the signature and input hash will be penalized with a lower σ_i as the deviation increases. Those matches whose deviation is greater than the one stored on the signature will get a zero similarity value.

The value of σ_i for a hash attribute is computed as follows:

$$\sigma_i = 1 - \frac{|h_i - h_{input}|}{deviation} \qquad \ldots (9)$$

where h_{input} is the hash of the input packet's payload, h_i represents the i-th hash in the signature, and *deviation* is the deviation attribute.

All other attributes are checked for exact matches. IP addresses, port numbers, window size, etc. are features for which different degrees of similarity do not work properly. If, for instance, a signature corresponds to an attack program that always targets a determined port number and the input uses a port number close to the port number in the signature (for instance,

port 22 and port 23), this does not mean the input in fact resembles the signature [103]. Different protocols may be run on different port numbers and so any similarity between signatures would be merely coincidental.

These attributes yield a 0 or a 1 that will enable or disable the attribute's weight in the calculation of σ. When the search reaches γ nodes, the input will be matched against multiple signatures under a determined β_i node. The highest σ score will be picked to denote to which signature the input resembles the most. The first-fit flaw [1] some intrusion detection systems suffer is avoided by comparing the input against a number of signatures and taking the closest match.

The following pseudocode provides a high-level perspective of how the matching algorithm for network signatures works:

```
# Network signature matching.  High-level pseudocode.
IF input MATCHES α_i node THEN
    ACCESS α_i list
    IF input MATCHES β_j node THEN
        ACCESS β_j list
        max = 0
        FOR all γ_k nodes IN β_j DO
            IF MATCH(input, γ_k) > MATCH(input, γ_max_i THEN
                max = k
            END_IF
        END_FOR
        PRINT The input corresponds to signature:  γ_max
    END_IF
ELSE
    PRINT The transfer is safe
END_IF
```

5.7 Matching process signatures

In the case of process signatures, we removed the *dynamic libraries* field considered in the design phase after reviewing the data. The number of dynamic libraries attack programs use is very small. This is perhaps due to the fact that these programs need to be portable and self-contained. A way of reducing platform dependency is to minimize the number of system libraries a program links at run time (malicious dynamic libraries are not included into the scope of this dissertation, however, they are used by some tools to compromise vulnerable systems [225, 154]). We found that more than 50% of the surveyed programs use at most two dynamic libraries. From the program modeling perspective this does not convey meaningful information to the signature and so has been omitted in the implementation.

The rest of the fields has been integrated into the signature. The text segment of Linux

[1] First-fit flaw. Returning the first match encountered during the search without considering subsequent, possibly better matches.

```
smurf4:     file format elf32-i386

Contents of section .text:

804885c 31ed5e89 e183e4f0 50545268 008f0408  1.^.....PTRh....
804886c 68d08e04 08515668 0c890408 e8fffeff  h....QVh........
804887c fff49090 5589e553 50e80000 00005b81  ....U..SP.....[.
804888c c3aa1a00 008b8378 00000085 c07402ff  .......x.....t..
804889c d08b5dfc c9c39090 5589e583 ec08803d  ..].....U......=
80488ac b8a30408 007529a1 08a20408 8b1085d2  .....u).........
80488bc 741789f6 83c004a3 08a20408 ffd2a108  t...............
80488cc a204088b 1085d275 ebc605b8 a3040801  .......u........
80488dc c9c389f6 5589e583 ec08a130 a3040885  ....U......0....
80488ec c07419b8 00000000 85c07410 83ec0c68  .t........t....h
80488fc 30a30408 e8fb76fb f783c410 c9c39090  0.....v.........
804890c 5589e556 5383c480 83e4f0b8 00000000  U..VS...........
804891c 29c4c745 c4000000 0083ec0c 68002000  )..E........h. .
804892c 00e8eafd ffff83c4 10894594 e8a60300  ..........E.....
804893c 0083ec08 683f8e04 086a02e8 a0fdffff  ....h?...j......
804894c 83c41083 7d08057f 1083ec0c 8b450cff  ....}........E..
804895c 30e89903 000083c4 1083ec0c 8b450c83  0...........E..
...
```

Figure 5.14: The text segment can be used to characterize a program. The figure shows an excerpt from the text segment of `smurf4`, one of the surveyed attack tools.

programs was chosen over the use of the heap and stack segments given its unchanging contents. Executable code is a valuable element for characterizing programs that is readily available with executables [211] (Figure 5.14). A change on this segment implies recompilation of the source program which can be limited by removing compilers from the system and enabling filesystem integrity checkers.

The number and order of arguments passed to a program are distinctive features highly dependent on the coding habits and preferences of the developer. File size is an attribute that completes the signature. It is used by integrity checkers in order to detect changes to critical files. To us it serves the purpose of confirming a hypothesis regarding signature matching.

When put together, all these information pieces help identify the type of program that is being executed. Even when some of these attributes can individually reveal the presence of an attack tool, there exist programs that remove themselves from the filesystem after execution [125] hereby eliminating the possibility of counting on the executable file resource to identify the attack program. It is for that reason that multiple attributes are considered in order to more effectively tell whether a certain rogue process is exploiting or surveying a system in the outside.

5.7.1 Primary data structure

For the matching of process signatures we use a linked list sorted on the *text segment size* attribute. The number of attributes composing process signatures is significantly less than the attributes used for network signatures. Each node in the process signature linked list represents an actual signature – notice that signatures corresponding to the same program may appear in nodes that do not share a link (Figure 5.15).

This list is permanently sorted and the matching algorithm checks first the node that has the closest text segment size to the input. It then proceeds to match the input against the predecessor and the successor of such node as explained in the next section.

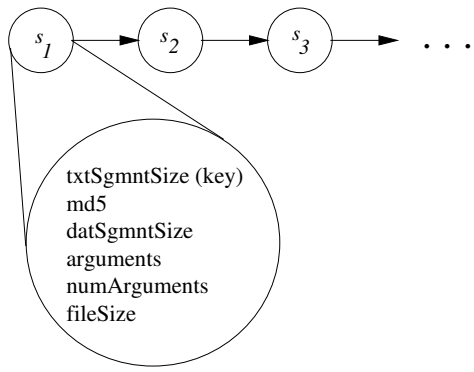


Figure 5.15: Process signature linked list. Every s_i node represents a program-variant signature.

The backup system used for network monitoring cells is also used for monitoring processes. Each process sensor announces to two backup peers – assigned in a circular fashion – its intention to analyze a certain program in execution. Unlike network sensors, no data capture is necessary. The only reference the monitor needs is the PID of the process. All other attributes can be derived from this reference. Should a sensor fail to report to its peers, the cascade sequence will activate one of its two peers which will take over the analysis of the program (see Figure 5.4).

5.7.2 Matching method for process signatures

A score system is also used for process signatures. An overall score π captures the similarity of the input with a signature node. Each signature attribute has a different weight ω_i that multiplies attribute sub-scores π_i:

$$\pi = \sum_{i=1}^{k} \pi_i \omega_i \qquad \ldots (10)$$

The following weights have been assigned to each attribute [184]:

- MD5: $\omega_1 = 0.2$. The matching function for the MD5 attribute returns a value $r_1 \in \{0, 1\}$ where 1 corresponds to an exact match and 0 to different hash values.
- Text segment size: $\omega_2 = 0.2$. The matching function returns a value $r_2 \in \{0, 1\}$ where 1 corresponds to an exact match and 0 to different text segments.
- File size: $\omega_3 = 0.2$. The matching function returns a value $r_3 \in \{0, 1\}$ where 1 corresponds to an exact match and 0 to different file sizes.
- Arguments: $\omega_4 = 0.2$. Depending on the number of arguments that match the signature, the matching function for this attribute returns a positive real value $r_4 \in [0, 1]$. Each argument contributes with $1/p$ to the value of π_4 where p is the maximum number of arguments that can be matched.

- Number of arguments: $\omega_5 = 0.1$. The number of arguments passed to the program at execution time contributes with $1/q$ where q is the maximum number of arguments the attack tool may receive.
- Data segment size: $\omega_6 = 0.1$. The matching function returns a value $r_6 \in \{0, 1\}$ where 1 corresponds to an exact match and 0 to different data segments.

The following is the pseudocode of the algorithm. It locates the most likely signature node using the text segment size field as an index and computes match score π with that node as well as with its predecessor and/or successor. The highest score of the three is chosen and is reported as the most likely match.

```
# Process signature matching.  High-level pseudocode.
tss = TEXT_SEGMENT_SIZE(input)
i = LOCATE_INDEX(tss, SignatureList)
IF i ≠ 0 THEN
    π1 = MATCH(input, SignatureList[i])
    IF SignatureList HAS PREDECESSOR
        π2 = MATCH(input, SignatureList[i].PREDECESSOR)
    ELSE
        σ2 = 0
    END_IF
    IF SignatureList HAS SUCCESSOR
        π3 = MATCH(input, SignatureList[i].SUCCESSOR)
    ELSE
        π3 = 0
    END_IF
    πmax = MAX(π1, π2, π3)
    PRINT The input corresponds to signature:  πmax
ELSE
    PRINT The process is safe
END_IF
```

5.7.3 Signature fusion

Monitoring cells process signatures separately. The results reported by a process monitoring cell are independent of the ones a network monitoring cell reports. If a network cell goes down and is unable to perform detection, or if an attack tools has been executed but the network interface is down and no packets have being generated, a process monitoring cell could still identify the program being executed in order for the security system to take action. Similarly, upon failure of a process monitoring cell, network cells can provide full detection support to the system by analyzing outbound traffic.

As proved by previous research, the integration of evidence sources is necessary to increase the accuracy of an intrusion detection system [1, 220]. This is especially important when automated response is being implemented. If what the system reports as illicit activity is really

violating the security policy in place, it is indeed possible to incorporate automated response into the security system. When detection results are not fully reliable, no effective automated response can be built on top of the intrusion detection system.

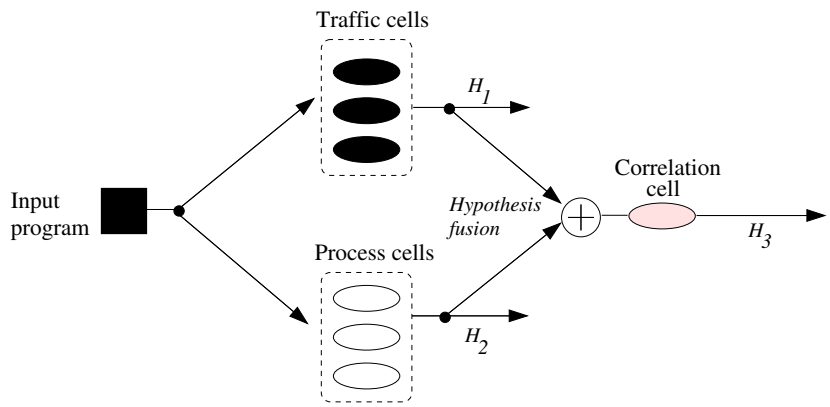


Figure 5.16: Fusing hypothesis H_1 and H_2 into H_3.

For that reason, the two detection systems implemented into FROID are integrated by correlation cells that determine a single score I which involves two hypotheses regarding which signature the observed input corresponds to. A **hypothesis** is the signature that the matching mechanism reports as most likely. In our implementation, hypothesis H_1 points at a certain network signature with similarity score σ, and hypothesis H_2 points at a process signature with similarity score π (Figure 5.16).

The integrated I similarity score is computed by a correlation cell that receives input from two monitoring cells as follows:

$$(1)\ If\ \{\exists\ H_1\ \wedge\ \nexists\ H_2\} \Rightarrow \{H_3 = H_1 \wedge I = \sigma\}$$
$$(2)\ If\ \{\nexists\ H_1\ \wedge\ \exists\ H_2\} \Rightarrow \{H_3 = H_2 \wedge I = \pi\}$$
$$(3)\ If\ \{\nexists\ H_1\ \wedge\ \nexists\ H_2\} \Rightarrow \{H_3 = I = NULL\}$$
$$(4)\ If\ \{\exists\ H_1, H_2 \mid H_1 = H_2\} \Rightarrow \{H_3 = H_1 = H_2 \wedge I = \tfrac{\sigma+\pi}{2}\}$$
$$(5)\ If\ \{\exists\ H_1, H_2 \mid H_1 \neq H_2\} \Rightarrow \begin{cases} \sigma > \pi & \Rightarrow \quad H_3 = H_1 \wedge I = \sigma \\ \pi > \sigma & \Rightarrow \quad H_3 = H_2 \wedge I = \pi \\ \sigma = \pi & \Rightarrow \quad H_3 = I = NULL \end{cases}$$

This cell will then implement automated response based on the information received from two monitoring cells and its reaction will be more definitive relying on the evidence found by independent detection mechanisms. The last section of this chapter describes response mechanisms.

5.8 Session tracing implementation

The session tracing model was implemented separately from the JADE agents at an earlier stage of the project [2]. Every agent is a stand-alone process that uses sockets to provide client and server functionality. It can request information from others and it can also share information.

A 1024-bit RSA key-pair $\{K_{Pub}, K_{Priv}\}$ is generated before the agent is launched. Once the agent is started up, it is ready to collaborate into the tracing of a session. When contacting a peer agent, it exchanges public keys and verifies the signature on the received key. The certification must come from the master key whose public key is accessible to all agents. In order to speed up message encryption, the sender follows a well known encryption procedure [179]: (1) it generates a session key K_{DES} for symmetric encryption; (2) it then encrypts the message using K_{DES}; (3) it encrypts K_{DES} using the public key of the recipient; and finally (4) it sends the encrypted session key and the encrypted message to the destination. The implementations of the RSA and DES algorithms we used are the ones provided by OpenSSL.

The prototype was developed on a RedHat Linux 8.0 box. Although agent libraries can be easily obtained for different compilers, they typically put unnecessary functionality into the agent that produces heavy processes. For this reason and with the intention of providing the best performance possible, the implementation was carried out in C minimizing the number of libraries linked to the executables.

Considering that programs and data files may get damaged in a security incident and that a security tool should not rely on them, our agents do not use the output from programs such as `finger`, `who` and `ps` to read the information stored on system tables like `utmp` and `wtmp`. They instead incorporate this functionality into their own body which gives them extended independence by making them self-contained entities. The size of each agent is around 20 kilobytes to which we add two encryption keys that need to be uploaded (this is just five additional kilobytes) along with the knowledge and rule bases.

The knowledge base of each agent includes a tree structure T where all connection chains are stored. Whenever a host has received all the information regarding a particular session, it stores the connection chain $c = [h_1 \rightarrow h_2 \rightarrow ... \rightarrow h_n]$ into T and sends it to all available agents. Every agent reads c and locates the hosts in c that match its own tree. It then creates new branches to keep its knowledge base up to date.

5.9 Reaction methods

Depending on the similarity scores reported, different response actions can be implemented. When the cells report high similarity scores, the type of reaction that can be performed can be more definitive. In case low scores are reported, it will be better to proceed safely by triggering actions that have not irreversible consequences.

Three cell types are capable of reaction: (1) network monitoring cells can react upon the packet sequences they observe, (2) process monitoring cells can react upon the execution of a process they consider as suspicious, and (3) correlation cells may act upon fused input to request session tracing and to alert neighboring systems.

[2] This prototype system was implemented and successfully tested; the integration with FROID cells is still pending (see Section 7.3).

The types of reaction implemented in FROID are a used to prove the need of having high detection-accuracy levels in order to be able to let a system automatically respond to activity classified illicit.

Response types

```
A = send matching data to evidence repository
B = move executable to evidence repository
C = copy command history to evidence repository
D = suspend process
E = change file execution privileges
F = send kill signal to process
G = start tracing
H = alert target system
```

Traffic Monitoring Cell			
	$\sigma \geq 0.5$	$\Rightarrow$	A
	$\sigma \geq 0.7$	$\Rightarrow$	B
	$\sigma \geq 0.8$	$\Rightarrow$	C
Process Monitoring Cell			
	$\pi \geq 0.5$	$\Rightarrow$	A
	$\pi \geq 0.7$	$\Rightarrow$	B
	$\pi \geq 0.8$	$\Rightarrow$	C
Correlation Cell			
	$\sigma \geq 0.5,\ \pi \geq 0.5,\ I\ \geq 0.7$	$\Rightarrow$	D
	$\sigma \geq 0.5,\ \pi \geq 0.5,\ I\ \geq 0.8$	$\Rightarrow$	E + G
	$\sigma \geq 0.5,\ \pi \geq 0.5,\ I\ \geq 0.9$	$\Rightarrow$	F + H

Table 5.6: Rules to implement automated response.

Notice that a single score can trigger more than one reaction. In the case of high similarity scores from monitoring cells, a correlation cell will integrate the hypotheses and will perform actions to effectively stop the attack launched toward an external system.

Chapter 6

Experiments and results

This chapter presents a set of experiments performed to test the implementation of the OID architecture. It describes the test set and experimental setup, the design of the experiments, and the results obtained from the evaluation of the accuracy and performance of the model.

6.1 Experimental objectives

The goal of the test phase is to analyze the feasibility of the OID approach and to evaluate the accuracy and performance of the FROID implementation under different traffic and workload conditions. The specific objectives are: (1) to evaluate the detection accuracy of individual cells, (2) to evaluate the detection accuracy of cells when detecting intrusion jointly, (3) to evaluate the performance of individual cells, and (4) to evaluate the performance of cells when detecting intrusion jointly.

6.2 Test data

The OID architecture has been tested using a sample group of automated remote-attack programs downloaded from the Internet. Some of these programs are proof-of-concept scripts that were developed to exemplify how a certain vulnerability can be exploited on a determined platform. They were obtained from vulnerability repositories and databases that store this kind of software and often provide references to security patches and workarounds [24, 149, 61].

The rest of the programs was obtained from the websites of different black-hat communities [150]. These programs have been specifically developed to exploit security bugs and they typically make reference to vulnerabilities already reported on vulnerability repositories.

6.2.1 Exploit types

The types of attack program being considered are four: (1) buffer-overflow exploits, (2) denial-of-service tools (DoS), (3) surveillance tools, and (4) misconfiguration exploits. Our search for experimental data confirmed the fact that buffer overflows are by far the most frequently exploited type of vulnerability and many of the tools surveyed belong to that type [124] (Table

Sophos Ex. 1007
Page 133 of 216

6.2). (Since we are evaluating the OID approach, we have only considered programs that target remotely-exploitable vulnerabilities).

DoS is a type of attack which is often automated through programs due to the need of submitting multiple requests or network packets to the target system in an attempt to rendering it inoperable [111]. Similarly, surveillance software often provides the possibility of analyzing a range of services, IP addresses, and port numbers. This is easily accomplished by means of scripts and programs that allow the user to repeat the execution of a series of operations. Misconfiguration errors are system administration flaws present at hosts with poor security management. The exploitation of this type of vulnerability has also been built into attack programs and therefore we have included some of them in our testing.

6.2.2 Target platforms

The programs that compose the test set can compromise the security of specific services and applications running on the target. In the case of buffer overflows, the pre-compiled code that has to be inserted into the RAM of the target system is platform-specific and some tools contain code for multiple platforms the intruder can choose from (Figure 6.1).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define DUMMY            0x41414141
#define BUFSIZE          512
#define IP_INDEX         5
#define PORT_INDEX       11

#define FMTSTR           0x1
#define OVERFLOW         0x2

#define JMPCODE          "\xeb\x0c"

static char linux_code[] =
    "\x31\xc0\x50\x50\x68\xc0\xa8\x01\x01\x66\x68\x30\x39\xb0\x02"
    "\x66\x50\x89\xe6\x6a\x06\x6a\x01\x6a\x02\x89\xe1\x31\xdb\x43"
    "\x30\xe4\xb0\x66\xcd\x80\x89\xc5\x6a\x10\x56\x55\x89\xe1\xb3"
    "\x03\xb0\x66\xcd\x80\x89\xeb\x31\xc9\x31\xc0\xb0\x3f\xcd\x80"
    "\x41\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80\x31\xd2\x52\x68\x2f"
    "\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xb0"
    "\x0b\xcd\x80\x31\xc0\x40\xcd\x80";

static char freebsd_code[] =
    "\x31\xc0\x50\x50\x68\xc0\xa8\x01\x01\x66\x68\x30\x39\xb4\x02"
    "\x66\x50\x89\xe2\x66\x31\xc0\x50\x40\x50\x40\x50\x50\x30\xe4"
    "\xb0\x61\xcd\x80\x89\xc7\x31\xc0\xb0\x10\x50\x52\x57\x50\xb0"
    "\x62\xcd\x80\x50\x57\x50\xb0\x5a\xcd\x80\xb0\x01\x50\x57\x50"
    "\x83\xc0\x59\xcd\x80\xb0\x02\x50\x57\x50\x83\xc0\x58\xcd\x80"
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
    "\x50\x53\x89\xe2\x50\x52\x53\x50\xb0\x3b\xcd\x80\x31\xc0\x40"
    "\x50\x50\xcd\x80";

struct target {

...
```

Figure 6.1: Sample attack program containing root-shell code for two different target platforms: Linux and FreeBSD.

Given their analysis purpose, network surveillance tools can be used to probe many types of system. A service may have been implemented for different platforms so the objective of surveillance programs will be identifying which of those implementations and what operating system the target system is running.

In order to reflect platform dependency, there are three platforms that are being considered: (1) Windows* systems (which includes all the MS Windows operating systems), (2) Unix/Linux (covering Unix and Linux variants), and (3) CISCO (includes CISCO 677/678, CISCO IOS, and Catalyst).

6.2.3 Programming languages

The bast majority of attack tools we found are written in any of two programming languages: C and Perl. Considering its support for pointer arithmetic, its predominant use in the development of Unix components and applications, as well as the functionality it provides for accessing system resources, C is the language of choice for many programmers that develop attack tools.

Unlike C, Perl is an interpreted language frequently used for creating shell and CGI scripts. It provides powerful data and text manipulation functions that make it suitable for writing vulnerability exploits for many services such as `HTTP`, DNS, and `Sendmail`.

We have considered these two languages in our testing to reflect the fact that they are the two most widely used languages for writing attack code.

6.2.4 Test program set

The experimental phase of the project considered fifty different remote attack programs. Their names appear listed in Table 6.2 and they cover the attack types, platforms, and languages mentioned in the previous subsections. Four of these tools require root access to be executed (`exceed, nxt, remo`, and `ss`). Again, OID covers the majority of the incidents under the assumption that inexpert users are not capable of hiding their activities. These four tools have been included into the testing set given the type of traffic they generate without considering the access level they require.

	Exploit name	**Type**	**Platform**	**Language**
1	305monit	Buffer overflow	Windows*	C
2	akill2	Buffer overflow	Windows*	Perl
3	anubisexp	Buffer overflow	Unix/Linux	C
4	apache-nose	Buffer overflow	Unix/Linux	C
5	apache-scalp	Buffer overflow	Unix/Linux	C
6	arksink	Buffer overflow	Unix/Linux	C
7	atphttp	Buffer overflow	Unix/Linux	C
8	bazooka	Buffer overflow	Windows*	Perl
9	bysin	DoS	Unix/Linux	C
10	cge	DoS	CISCO	Perl
11	cobain	Buffer overflow	Unix/Linux	C
12	delefate	Buffer overflow	Windows*	C
13	emule	DoS	Windows*	Perl

14	esignal	Buffer overflow	Windows*	Perl
15	ethboom	DoS	Windows*-Unix/Linux	C
16	exceed	DoS	Unix/Linux	C
15	ethboom	DoS	Windows*-Unix/Linux	C
16	exceed	DoS	Unix/Linux	C
17	eXtremail2	DoS	Unix/Linux	C
18	ezploit	Misconfiguration	Windows*	C
19	fping	Surveillance	Windows*-Unix/Linux	C
20	freezer	DoS	Unix/Linux	C
21	fspscan	Surveillance	Windows*-Unix/Linux	C
22	gtkftpd	Buffer overflow	Unix/Linux	C
23	iawebmail	Buffer overflow	Windows*	Perl
24	ircd	DoS	Windows*	C
25	isspam	Buffer overflow	Windows*	C
26	lsasrv	Buffer overflow	Windows*	C
27	namedsploit	Buffer overflow	Unix/Linux	C
28	niprint	Buffer overflow	Windows*	C
29	nptd	Buffer overflow	Unix/Linux	C
30	numby	Surveillance	Unix/Linux	C
31	nxt	Buffer overflow	Windows*-Unix/Linux	C
23	pam	Buffer overflow	Unix/Linux	C
24	phorum	DoS	Windows*	Perl
25	picobof	Buffer overflow	Windows*-Unix/Linux	C
26	priv8	Buffer overflow	Unix/Linux	Perl
36	proftpd	Buffer overflow	Unix/Linux	C
37	qex	Buffer overflow	Unix/Linux	C
38	realown	Buffer overflow	Windows*	C
39	remo	Buffer overflow	Windows*	C
40	rootdown	Misconfiguration	Unix/Linux	C
41	rpcexec	Buffer overflow	Windows*	C
42	smurf4	DoS	Windows*-Unix/Linux	C
43	spxeneo	DoS	Windows*	Perl
44	ss	Surveillance	Windows*-Unix/Linux	Perl
45	sslbomb	DoS	Windows*	C
46	symanhoddos	DoS	Windows*	C
47	telexp	Buffer overflow	Windows*	C
48	tlsdos	DoS	Windows*	C
49	vwexploit	Buffer overflow	Unix/Linux	C
50	yapping	Surveillance	Windows*-Unix/Linux	C

Table 6.2: Automated remote-attack tools used in the experiments.

6.3 Experimental setup

In order to run the experiments we used a setup that included two source systems (*attackers*) and three target systems (*victims*). These two groups were connected to separate segments of a LAN that supports hundreds of users every day [1].

Name	Function	OS	RAM	Processor	NIC
System 1	Source	RedHat Linux 9.0	512Mb	Intel Celeron @ 800MHz	10/100 Ethernet
System 2	Source	RedHat Linux 9.0	1Gb	Intel Pentium @ 1.8GHz	10/100 Ethernet
System 3	Target	MS Windows 95	128Mb	Intel Pentium @ 700MHz	10/100 Ethernet
System 4	Target	MS Windows 98	512Mb	Intel Pentium @ 1.2GHz	10/100 Ethernet
System 5	Target	RedHat Linux 7.0	256Mb	Intel Pentium @ 650MHz	10/100 Ethernet

Figure 6.2 depicts our experimental setup. The two source machines run RedHat Linux 9.0 and each of the target machines supports a different operating system (Windows 95, Windows 98, and RedHat Linux 7.0). The targets were all configured with default settings from the installation CD's and no service packs or patches were applied. The three target systems run different operating systems that have services vulnerable to a good number of existing attacks. The selected programs exploit flaws on TCP/IP-based network services that are installed along with the operating systems (for instance, service daemons for HTTP, DNS, News, e-mail, and network printing; all these are services reachable through the network).

The OID approach works with one or more machines that try to defeat the security of a target using the Internet and a suite of attack and surveillance tools. Although instances of FROID can be installed on all the machines, OID focuses on detection at the source of the event and there is where FROID agents were installed.

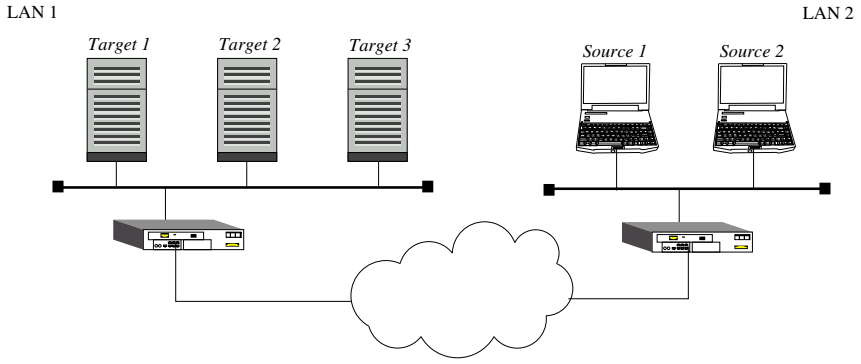


Figure 6.2: Experimental setup for the testing phase.

[1] The CSF network of the Department of Computer Science of Cornell University as well as resources from the Center for Intelligent Systems and the Department of Computer Science of ITESM were used in the development and testing of FROID.

6.4 Signature generation

Source system 1 was used to generate signature pairs for each of the 50 attack programs (process signatures appear in Appendix E and network signatures in Appendix F). Two cells need to be active when a program is to be modeled: a network signature cell and a process signature cell.

The system is executed via the Java compiler which receives a call to the JADE environment class:

```
% java jade.Boot N:NetworkSignatureCell T:TrafficSignatureCell
```

Once active, both cells will download the ontology and will start polling the connections file for new entries that indicate a program is sending data through the network interface. They will separately generate signatures that will be appended to the signature repositories of processes and network traffic. For the purpose of having a controlled environment, no network activity other than the one created by the program in evaluation was present at signature-generation time.

For each program at least one execution variant was considered. In order to further classify variants, we divided the behavior of the target system into *vulnerable* and *not vulnerable*. From a networking stand-point, the reaction the source sees from its target differs between these two categories and for that reason, each of the signatures belongs in one of these two groups. As a result, attack attempts as well as successful attacks can be modeled and detected.

The systems takes in average 2.44 sec to compute a network signature and 1.20 sec to compute a process signature. Network signatures take more time since in some cases packets are delayed and the sensor of the monitoring cells has a behavior that makes it wait until a certain number of packets have been received or until the 6 sec timeout threshold is reached; whatever comes first.

After signature generation, signature matching experiments took place.

6.5 A word on compiled and interpreted code

Our test set included 10 interpreted-code programs (PERL programs) and 40 compiled-code programs (C programs). Network-level detection works the same in both cases: a stream of network datagrams is generated by the executable or by the interpreter and it is analyzed by monitoring cells for signs of outbound intrusion.

For process-level detection, the text segment of the interpreter includes the code of the program to execute. This limits the task of computing a hash of a text segment for characterizing the program. However, arguments, file size, etc. are available for evaluating the similarity of the input program with the signatures in the repository. For keeping the scoring system consistent, all PERL programs received a π_i sub-score of $0.2/2 \times 1$ for the MD5 attribute – otherwise all PERL programs would have gotten lower process-signature scores.

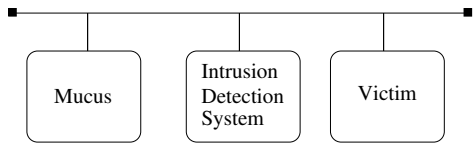
6.6 Experiments: Signature matching and correlation

The signature matching method was tested with accuracy and performance experiments. The two main factors that influence these evaluations are the system's internal workload and the

amount of traffic that goes through the network interface [184]. These variables should be controlled in order to effectively measure the precision and response time of the system.

6.6.1 Traffic generation

In order to control and measure the amount of traffic in our tests, we used Mucus [136]. Mucus is an open-source traffic generation tool designed to evaluate network-based intrusion detection systems. It translates Snort signatures [170] into network packet streams and allows the user to vary transmission rates, to modify their contents and type, and to randomize payloads (Figure 6.3). Mucus can be run on any computer that has a network connection.



(a) Topology

```
[smv@localhost mucus-1.0]# mucus
usage: mucus
  [-n <number-of-attack-instances>]
  [-r <random-seed>]
  [-S] (suppress-network-traffic)
  [-d <destination-ip-address>]
  [-s <source-netmask>]
  [-D <fixed-delay-in-seconds>]
  [-v] (turn on verbosity)
  [-R] (random 0.1 - 1.0 sec delay
  [-A] (random rule selection)
  [-N <rule-number>] specific rule
  [-P] (randomize packet payloads, default)
  [-p] (don't randomize packet payloads)
  [-E] (establish TCP sessions for TCP rules)
    [-G <fake-gateway-ip>] (required by -E)
    [-V <fake-victim-ip>]  (required by -E)
  [-e] (don't establish TCP sessions, default)
  ** snort rules are expected on the standard input
```

(b) Options

```
[smv@localhost src]# ./mucus -n 10 -A -r 1234 -d 192.168.1.20 -s 128.0.0.0/8 < ./snort-rules-186
--[ MUCUS options ]---------------
number of attack instances: 10
random seed: 1234
no fixed delay specified
using fixed destination address: 192.168.1.20
randomizing packet payloads.
not establishing TCP sessions.
  ** generating network traffic **
----------------------------------
HOME_NET == 192.168.1.0
EXTERNAL_NET == 0.0.0.0
SMTP == 192.168.1.0
HTTP_SERVERS == 192.168.1.0
SQL_SERVERS == 192.168.1.0
DNS_SERVERS == 192.168.1.0
initializing RNG with seed: 1234
1225 rules parsed.
10 instances of these rules will be generated.
[Wed Mar 16 12:11:08 2004 378] TCP - "WEB-CGI AT-admin.cgi access" - 128.197.61.87:38350 -> 192.168.1.20:80
[Wed Mar 16 12:11:08 2004 381] TCP - "EXPLOIT NNTP Cassandra Overflow" - 128.232.59.235:56184 -> 192.168.1.20:119
[Wed Mar 16 12:11:08 2004 383] TCP - "INFO - Possible Squid Scan" - 128.219.40.5:237 -> 192.168.1.20:3128
[Wed Mar 16 12:11:08 2004 388] ICMP - "ICMP Source Quench (Undefined Code!)" - 128.104.184.71:51294 -> 192.168.1.
...
```

(c) Execution example

Figure 6.3: The Mucus intrusion detection stimulator.

Mucus version 1.0 had to be modified for it to accept shorter packet delays. By default, delays can only go from 0.1 to 1 sec, which is not a good granularity for simulating high traffic loads. The `usleep()` function used in the program receives an argument in microseconds which is multiplied by 1×10E6. We changed this in order to allow the traffic generator to receive delays in microseconds as well. The change in `mucus.cpp` (lines 498–502), was :

```
// before returning, sleep some time as specified by the user
if (_fixedDelaySpecified) {
  //usleep((unsigned int) ((float) _fixedDelay * (float) 1E6));
  usleep((unsigned int) ((float) _fixedDelay * (float) 1));
}
```

Three traffic workload levels have been defined for our testing (the effect of the system load on Mucus is considered part of the normal functioning of the system under those conditions and has not been separately analyzed; the generation of packets for purposes of OID cannot be isolated to a different box). They were used to create different execution conditions at the system:

1. **Low traffic volume**. No network packets are artificially generated. The only traffic present is the one produced by the attack tool itself.

2. **Moderate traffic volume**. An incoming packet is generated every t_i microseconds where $t_i \in [1, 500]$ is a pseudo-random integer, and an outgoing packet is generated after t_o microseconds of the incoming packet where $t_o \in [1, 500]$ is a pseudo-random integer.

3. **High traffic volume**. An incoming packet is generated every t_i microseconds where $t_i \in [1, 10]$ is a pseudo-random integer, and an outgoing packet is generated after t_o microseconds of the incoming packet where $t_o \in [1, 10]$ is a pseudo-random integer.

Sensor agents see all the packets the network interface handles – that includes incoming as well as outgoing traffic. They select outgoing packets and pass them on to correlation agents within the same cell. For that reason, the experiments consider both, artificially-generated incoming and outgoing traffic.

6.6.2 Workload control

The workload at the source system has an impact on the performance of agent cells. When many applications are executing concurrently, cells will have to compete for resources through the scheduling mechanism.

In order to test our application under workload conditions we could replicate, we define two levels:

1. **Low workload**. At low workload the source system does not run any application other than the operating system, the monitoring cells (i.e., Java virtual machines), and the program that is to be observed.

2. **Moderate workload**. In order to generate a moderately high level of system activity consistent throughout the testing, we executed a set of applications on the source machine. These programs consume a high volume of processing cycles per execution and were run in parallel to our JADE cells.

   Three programs were selected and they were executed twice each in order to spawn a total of six processes. All programs, including JADE cells, were executed with root privileges. The programs were: (1) `find` (searches the hard disk for a file; the system takes around 17 sec to

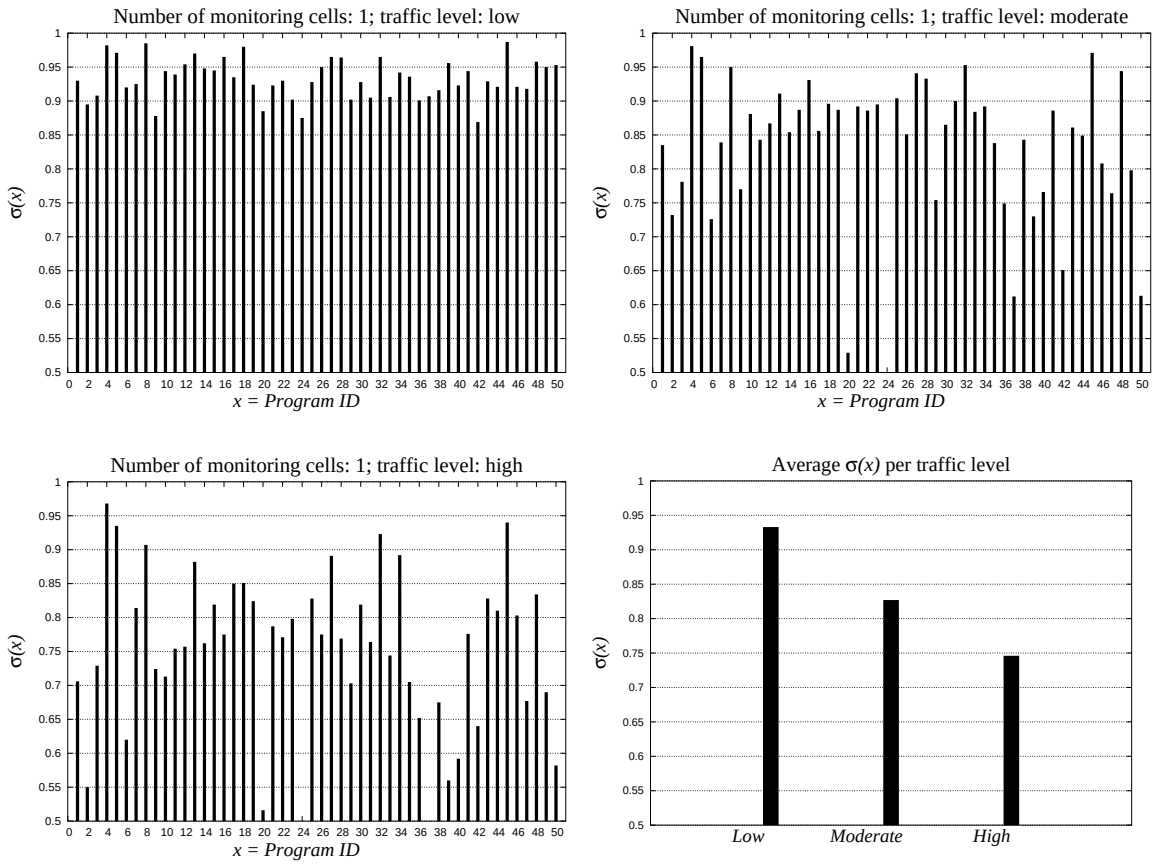


Figure 6.4: Average hypothesis score $\sigma(x)$ corresponding to each attack tool using one traffic-monitoring cell.

locate the file), (2) `top` (is a permanent loop that lists the processes executing on the server in real time), and (3) `cp` (copies a 94 Mb file from one disk partition into another; the copy takes around 12 sec to complete).

6.6.3 Cell accuracy

In order to evaluate the accuracy of the system several experiments were performed using *source system 1* and the three targets:

1. Accuracy of traffic-monitoring cells operating independently. We evaluated the ability of traffic-monitoring cells to detect attack programs in execution.

We run three experiments to include one, two, and three traffic-monitoring cells active at the same time. For each case we considered three sub-cases corresponding to each of the three traffic levels defined in previous sections.

The 50 attack programs were executed three times each. All the signatures generated during the signature generation phase were present on the signature repositories of the cells.

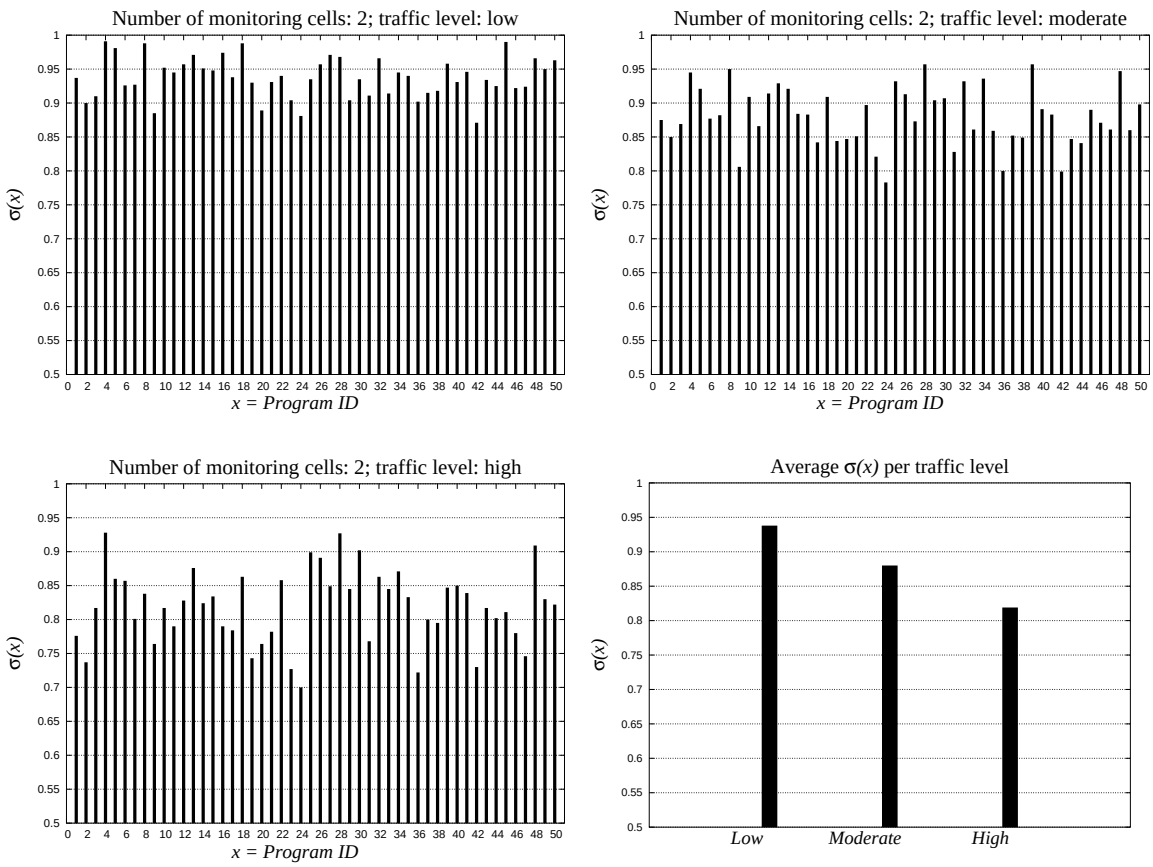


Figure 6.5: Average hypothesis score $\sigma(x)$ corresponding to each attack tool using two traffic-monitoring cells.

We computed the average hypothesis score σ from the three executions of each program. The results for one, two, and three monitoring cells are shown in Figures 6.4, 6.5, and 6.6.

The precision of the system varies with the number of active cells. When only one cell is active all matching scores lay above 0.8. However, even with moderate traffic load, the system experiences a clear decrease in its ability to positively detect the programs in execution (from an average of 0.931, scores drop to 0.827 for a 11.1% accuracy loss). A single cell has no backup peers that support its execution. A missed packet in a transfer may not be recovered and this makes the matching method produce a low score as it is unable to verify all the packets necessary to match certain signatures on the linked list. At high traffic volume, a lowest score 0.479 appears and the average σ drops 9.69% to 0.748. This average could be acceptable but the system fails to detect eight attack programs with scores below 0.6 where automated response could not safely take place.

When one more cell is added, accuracy increases. Accuracy under low traffic volume remains stable and high, but there is an improvement at moderate and high traffic volumes. When two cells are active, each one functions as the read peer of the other. Packets are evenly split between them and more resources are available for the processing. At moderate volume, an average score of 0.883 is achieved and 0.820 at high volume traffic. These represent an improvement of 6.77%

and 9.65% for moderate and high volume traffic respectively, from the execution with only one network-monitoring cell.

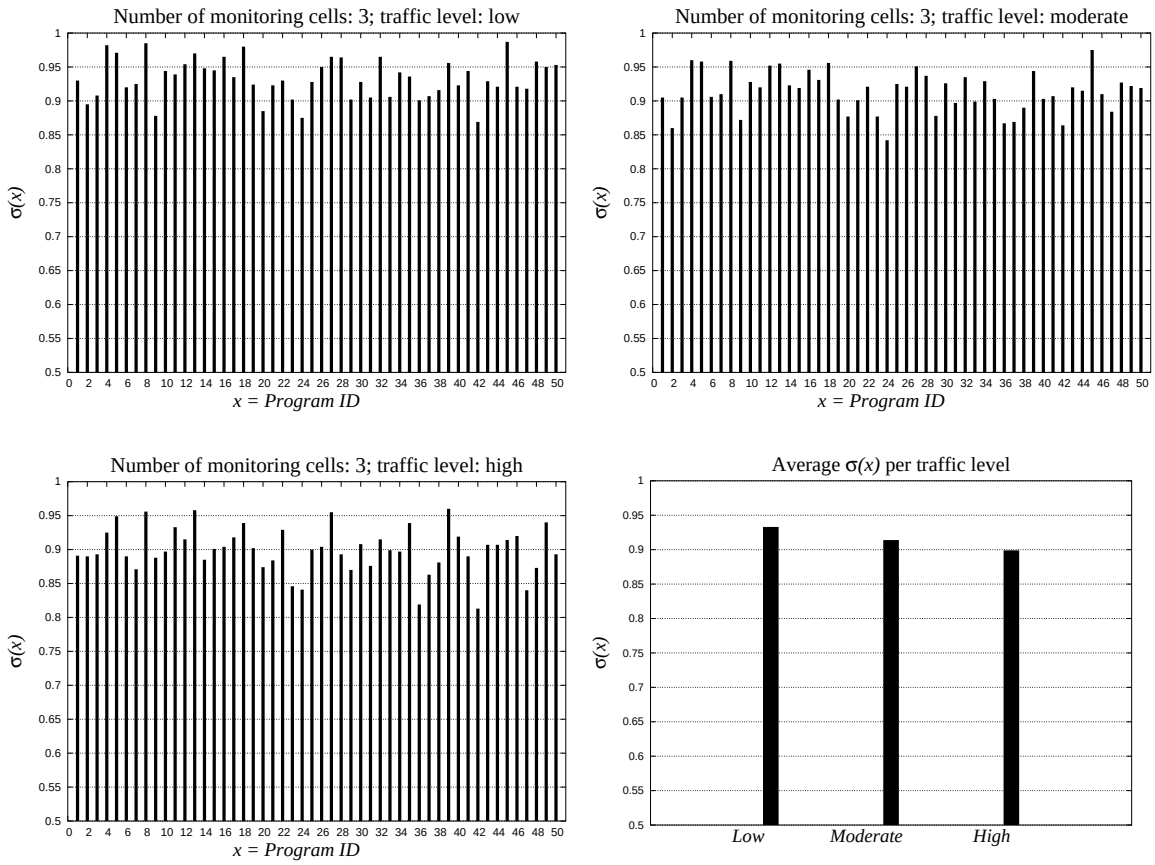


Figure 6.6: Average hypothesis score $\sigma(x)$ corresponding to each attack tool using three traffic-monitoring cells.

		Number of cells		
		1	2	3
Traffic	Low	0.931	0.941	0.933
	Moderate	0.827	0.883	0.921
	High	0.748	0.820	0.895

Table 6.4: Matching scores for one, two, and three traffic-monitoring-cell scenarios.

The execution with three monitoring cells renders the system slow but detection accuracy clearly improves. Low traffic still yields high scores and accuracy at moderate and high traffic conditions goes up to 0.921 and 0.895 respectively. This accounts for a 4.30% increase and 9.14%

from the two-cell case, and 11.36% and 19.65% from the one-cell case for moderate and high volume traffic, respectively.

Monitoring network traffic with more than one cell gives better detection results at the cost of performance (see next section for performance evaluation results).

2. Accuracy of process-monitoring cells operating independently. We evaluated the accuracy of process-monitoring cells when they operate independently (i.e., no correlation). We considered scenarios with one and two cells. A three-cell scenario was not considered given that, unlike network packets that need to be buffered for analysis, program execution information is always placed on more stable storage (the `/proc` pseudo-filesystem, e.g.) and this helps cells that missed to capture information pieces belonging to their assigned process. The cell backup system is still in place in case case the cell is unable to monitor a process.

Each of the test cases has two variants corresponding to different workload levels. Figures 6.7 and 6.8 show average scores from three executions per program.

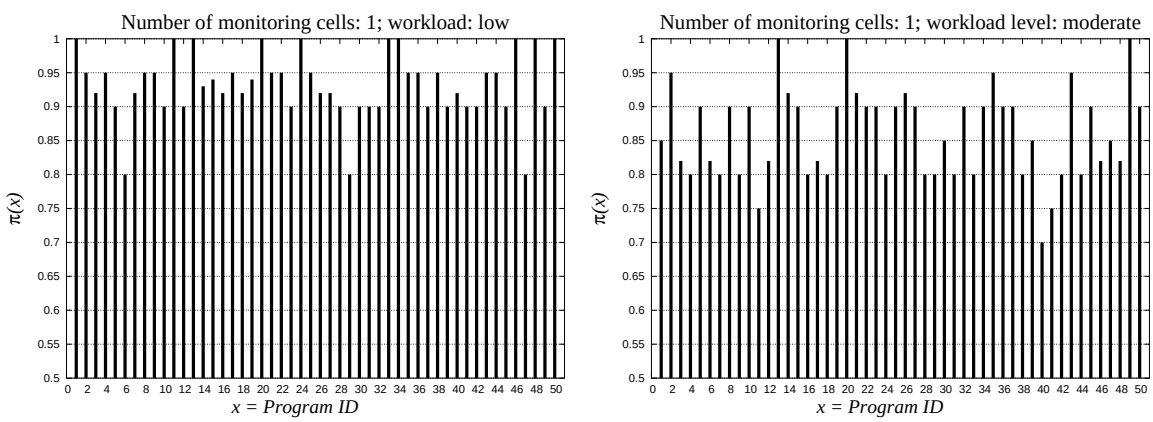


Figure 6.7: Average hypothesis score $\pi(x)$ corresponding to each attack tool using one process-monitoring cell.

When a single traffic-monitoring cell is active, detection accuracy is over 0.75 for the 50 programs when they execute under low workload (remember that scores are dependent upon the individual variants of the program). A 0.8 lowest hypothesis score is achieved for program ID's 6, 29, and 47 (see Table 6.2) and many other programs are identified with a 1.0 score. Score differences are larger than the those obtained with network cells because there are less attributes and most of them return a boolean value that is multiplied by a weight $\omega \in \{0.1, 0.2\}$.

When we execute the six processes for generating a heavier workload, accuracy decreases – although we still have a few programs scoring 1.0. If the program is not successful at compromising its target, execution will complete in a few milliseconds and its reference data on `/proc` will disappear. In that case, not all the attributes of the signature will be checked and the score will be low.

Having two process-monitoring cells does not significantly improve the accuracy of the system when operating under low workload. The fact that programs may fail to compromise

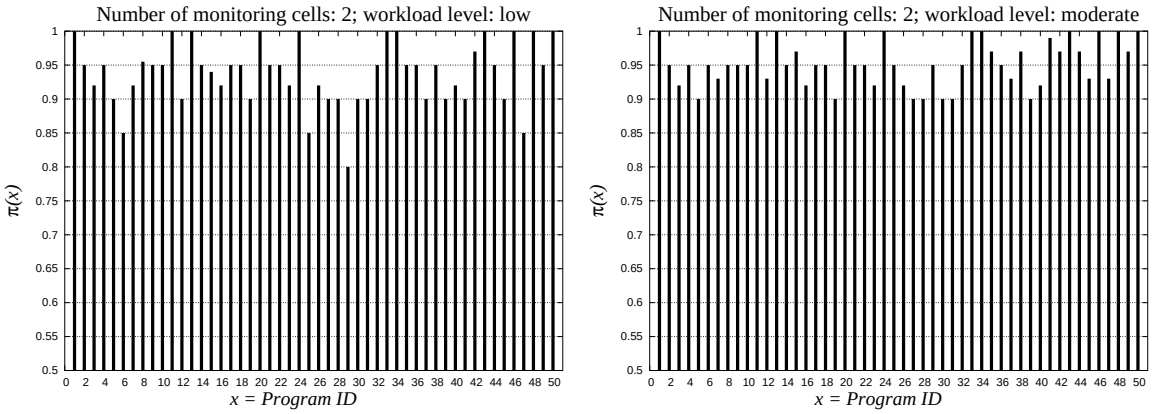


Figure 6.8: Average hypothesis score $\pi(x)$ corresponding to each attack tool using two process-monitoring cells.

a target and will therefore execute very quickly equally affects one or two cells. Two cells are able to handle moderate workload better, though. This is most likely because, while the cell to which the program was assigned is busy catching and analyzing data, the backup cell is ready to jump and take over the checking. The smallest score obtained with low workload is 0.8 whereas with two parallel cells the lowest score is much higher at 0.9.

6.6.4 Cell performance

The time FROID cells take to detect the execution of a program varies with the number of active cells, the workload at the source system, and the amount of network traffic. Average detection times from three executions of each remote automated-attack tool were computed and are shown next.

1. Detection time with network monitoring cells operating independently. Figure 6.9 contains the detection times of network monitoring cells. Each program was executed three times and an overall average of all of them was computed for the three levels of network traffic previously defined.

The execution of cells using an interpreter or, in this case, a virtual machine, is very expensive performance-wise. The benefits of portability and isolation the cell concept provides have a cost in performance. This is important as an critical aspect for intrusion detection tools is real-time detection. In a matter of milliseconds a system may be compromised and that is why fast, effective detection and correlation algorithms are necessary.

When one cell executes under low traffic conditions, detection takes 1.427 sec in average. This time is high compared to other intrusion detection systems that report 0.6 sec or less to detect an intrusion [103, 224]. A comparison like this, however, lacks validity as the intrusion detection approach is different from the one followed by OID. OID will detect programs in execution on our host. The evidence captured by cells gives the system more time to evaluate hypotheses and the fact that no packets or programs are blocked until the detection mechanism yields an outcome – as some intrusion detection tools do [8, 188, 164] – allows FROID to take

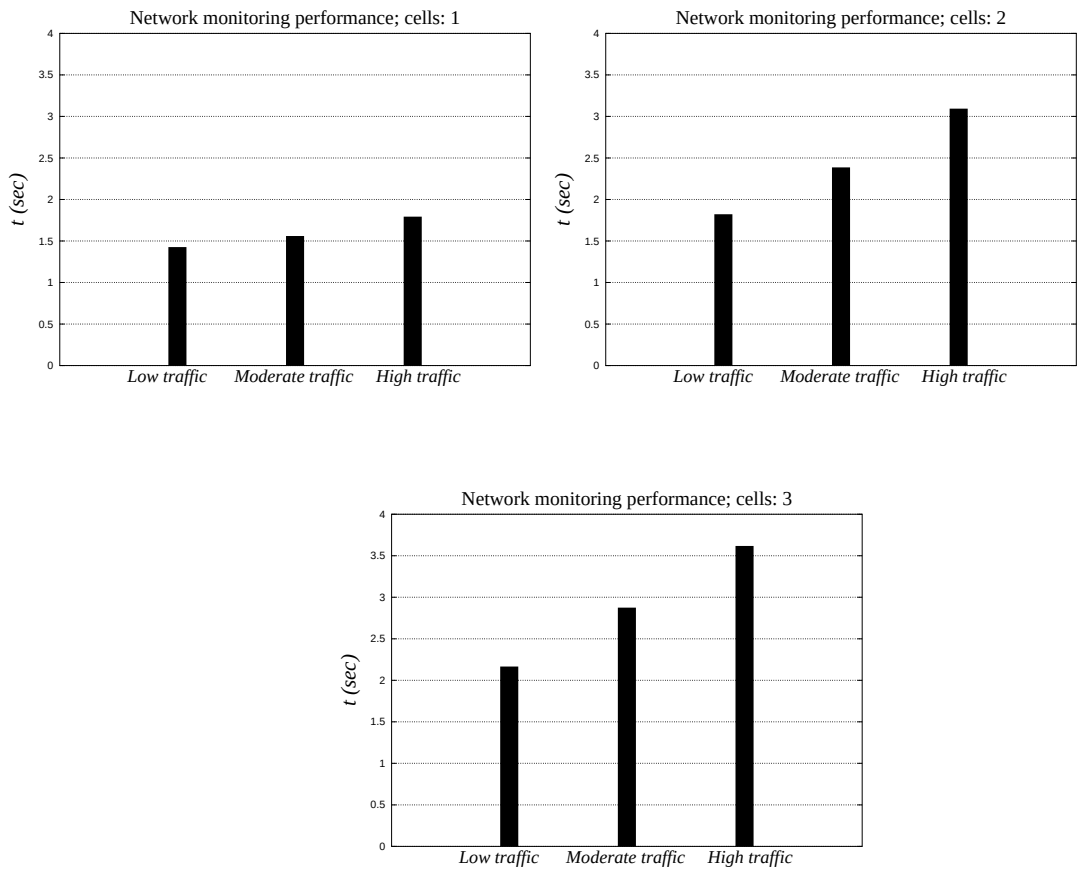


Figure 6.9: Detection time with network monitoring under different traffic levels using one, two, and three cells.

more time and still be able to identify the tools.

With low traffic, a network cell takes 1.560 sec in average to detect an attack tool and 1.797 sec when it operates under high traffic volume. These increments are acceptable. When working with low traffic, packets arrive one after the other with no major delay. Under heavier traffic conditions, the system will have to actually do filtering and this will increase detection time.

The execution of a second cell implies an additional virtual machine which means slower detection time (for accuracy figures please refer to the previous section). Time almost doubles from low traffic with one cell (1.427 sec) to high traffic with two cells (3.084 sec). The issue with such a detection time is that an attack tool or the attacker himself could have the time to remove evidence from the filesystem, memory, and even `/proc` if the security mechanism takes even few seconds to respond.

The last case with three cells considerably increases detection time to 2.166 sec with low traffic, 2.875 sec with moderate traffic, and 3.618 sec under high traffic conditions.

2. Detection time with process monitoring cells operating independently. A similar increase is observed when more than one process monitoring cell is active. Figure 6.10 contains

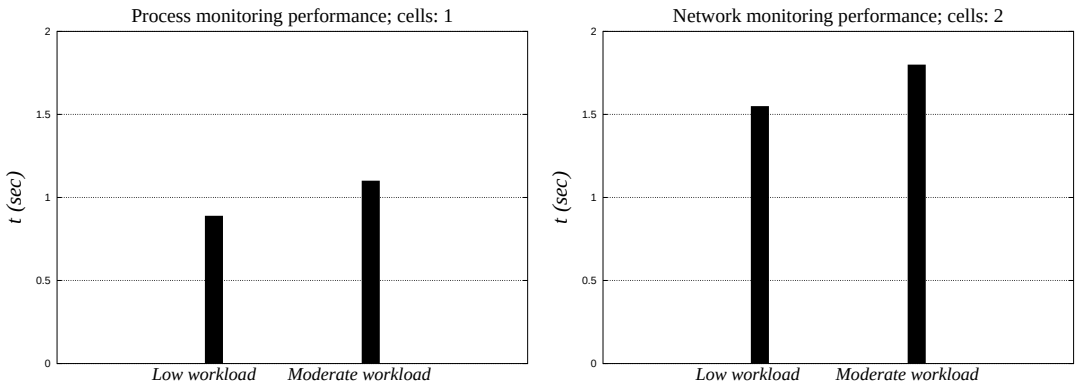


Figure 6.10: Detection time with process monitoring under different workload levels using one and two cells.

average detection times for all the 50 attack programs executed three times each and checked by one and two cells.

The impact of adding a second cell is a 79.64% increase in detection time under low workload conditions (from 0.894 sec it goes to 1.606 sec) and 60.69% under moderate workload (from 1.183 to 1.901 sec).

Agents are executed as threads within cells but their performance is not as relevant as observing the behavior of the system when multiple virtual machines are running carrying out intrusion detection tasks. This is because the proposed architecture involves software sub-environments and a way to implement them is through virtual machines and interpreters.

All the above charts expose the high cost of implementing agent cells through Java virtual machines. The constructs JADE provides are a convenient way to implement a multiagent platform but using more than one environment at a time has an prohibitively high cost that affects the performance of the source system where rogue programs are supposed to be detected.

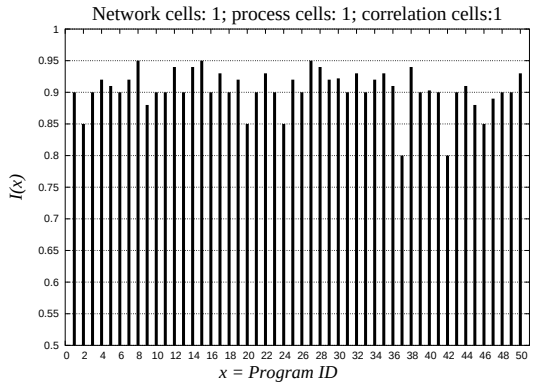


Figure 6.11: Detection accuracy using correlation with one network-monitoring cell, one process-monitoring cell, and one correlation cell.

6.6.5 Correlation accuracy

The correlation method described in the previous chapter was tested for performance and accuracy. The latter involved combining the outcome from each monitoring cell in charge of measuring the similarity of the program's behavior with the signatures.

In order to compute hypothesis scores (I scores) two experiments were executed. The first one involved one network-monitoring cell and one process monitoring cell executing on the same source system with low traffic and workload conditions (see Sections 6.6.1 and 6.6.2 for details). The second one employed two concurrent network-monitoring cells and two process-monitoring cells working under high network traffic conditions.

1. One network-monitoring cell, one process-monitoring cell, and one correlation cell; low network traffic; low workload. All process and network signatures were uploaded into both cells. Mucus was not used and no other applications were executing on the source system. Both cells were reporting an outcome to the correlation cell.

The two cells were spawned one after the other and each of the 50 attack programs was executed once. Figure 6.11 shows the I scores reported by the correlation cell.

Since the two monitoring cells are reporting a hypothesis and a score (i.e., no cell failures) we are able to integrate the scores from the two matching systems without having to consider the cases where one of the hypothesis is missing. In order to trigger reaction, an I score must have to be at least 0.7 to suspend the process corresponding to the PID the cells received, and both σ and π scores must exist and be 0.5 or greater. The certainty that two cells are reporting the same hypothesis allows for the cell to implement automated response based on sound evidence.

Average correlation scores are smaller than network-monitoring scores but they have the big advantage of having been generated using independent entities that evaluate the input program separately.

2. Two network-monitoring cells, two process-monitoring cells, and two correlation cells; high network traffic; moderate workload. Mucus was set up to generate a high amount of traffic that kept the network interface rather busy and two instances of each of the three application selected for producing a moderately high system workload were launched. These six processes were accompanied by six FROID cells: two for network monitoring, two for process monitoring, and two for correlation. Figure 6.12 shows the average I scores computed between the two correlation cells for each attack program.

Average scores are much lower than when single cells were used. This is most likely due to the amount of traffic being generated and the time each cell has for evaluating each program. All I scores range from over 0.7 to 0.971 which is enough for triggering automated reaction.

An important point to make is that the workload and traffic conditions of this experiment really appeared as an abnormally heavy scenario at the source system that allowed us to test the functioning of the architecture. Despite of this, the accuracy level permitted the system to positively identify the programs in execution that had a signature on any of the repositories.

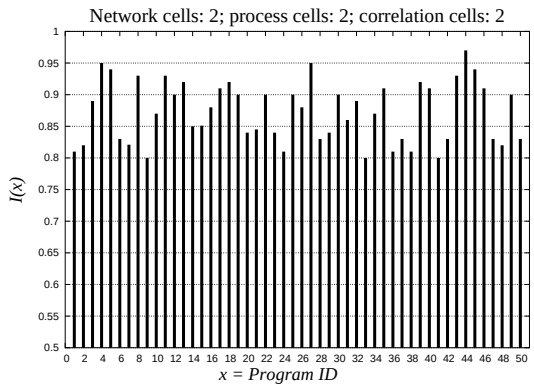


Figure 6.12: Detection accuracy using correlation with two network-monitoring cells, two process-monitoring cells, and two correlation cells.

6.6.6 Correlation performance

The two experiments involving correlation exhibited poor performance. The use of several virtual machines (up to six) proves not to be an effective way of implementing agent cells. Figure 6.13 shows the average time that correlation cells took to produce an I score for the input – this represents detection time for single executions of all the programs corresponding to the two experiment scenarios exposed in the previous section.

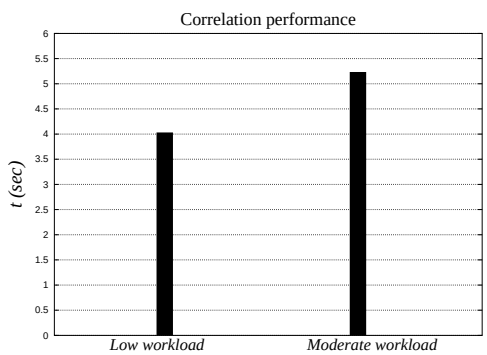


Figure 6.13: Left: average detection time using two monitoring cells and one correlation cell. Right: average detection time using two pairs of monitoring cells and two correlation cells.

A worst case of 5.23 sec detection time occurs when six cells are executed. This time does not represent an issue from the accuracy perspective, but giving a rogue program five execution seconds could have severe consequences on the target. Furthermore, attack systems could be small or big servers. A big, corporate server could probably host several instances of the FROID cells to provide continuous checking, however, a small system running the software would have trouble delivering fair performance to the user, which is one of the important challenges of intrusion detection technology.

6.7 Interactive session tracing

In order to evaluate the efficiency of the tracing method, some experiments were prepared. The testing environment consisted of set of seven Linux machines running kernels 2.1 and above on RedHat 7.2, RedHat 7.3, RedHat 8.0 and Mandrake 8.0 operating system installations.

The general structure of the experiments was the creation of connection chains of length $m \in \{2, 4, 8, 16, 32\}$ using n hosts where $n \in \{3, 5, 7\}$. This number of test boxes was selected since our experience tells us it is not likely that, for session performance purposes, an attacker – especially beginners – can use more than seven stepping-stone hosts.

All cases where there is no loop in the chain, that is, where $m = n - 1$, are easily resolved so we prepared connections that include a series of loops along the chain. For instance, in the three-server scenario, the connection goes in circles from host h_1 into h_2, from host h_2 into host h_3, from host h_3 back into host h_1 and so on for a total of m hops. The experiments show that the algorithm effectively unwinds the loops. When an agent finds the same loop repeated several times, it will not try to solve it over and over again. Its knowledge base will contain user and connection information that allows it to conclude that a user is creating a loop.

Part of the contribution of this method is the fact that, by using loop unwinding, the performance of the algorithm has an upper threshold which is linear on the number of hosts n through which the user connects. A set of n agents will be in charge of tracing the connection and even for a very long chain, an agent located on each hosts guarantees that loops will not delay the tracing process in any way.

m	Time without encryption (sec)	Time with encryption (sec)
2	0.03	0.84
4	0.10	1.13
8	0.62	3.09
16	0.69	4.20
32	0.73	3.99

Table 6.5: Tracing time with and without encryption (RSA key exchange + DES chaining block ciphering); m corresponds to the number of connection hops or length of the chain.

Table 6.5 includes tracing times with and without encryption. Both columns indicate that the growth of the tracing time is below linear with respect to the number of hops m (this makes us think this model should escalate well to large networks). The time difference from tracing with and without encryption is considerable. This only reminds us of the performance cost of using cryptographic algorithms to protect our data. It is not likely that a connection chain extends over more than thirty hosts, for instance (latency would render the connection inoperable). It follows that, even with 1024-bit keys, the use of encrypted communication is the way to go regarding agent communication for this problem.

In all cases the system was able to locate the origin of the connection. However, no experiments were performed as to evaluate the robustness of the model once one or more agents stop

cooperating.

6.8 Results

The results of the test phase indicate the multiagent approach to OID is feasible when using agent and ontology technologies. The performance of the FROID prototype, however, reveals clear performance limitations when more than one cell executes on the host system. Remote-attack programs can be detected by agents that characterize them as processes and also as network traffic generators. By integrating both detections an agent is able to provide a more accurate identification of the executing program. Grouping agents into teams provides the protection mechanism with increased robustness measured by the accuracy of the detection when workload and traffic conditions increase and the backup mechanism gets activated (misses remain low). The definition of program signatures allows the system to have an ontology-supported signature repository that can be deployed over the network in order to minimize exposure while speeding up reaction time.

Chapter 7

Conclusions, contributions and future work

This chapter condenses final remarks and conclusions, lists the contributions of the dissertation, and describes potential lines for future research.

7.1 Conclusions

- Monitoring for compromise attempts launched locally toward external targets is an effective way of countering the expansion of security threats.
- Outbound intrusion detection is a feasible approach to systems monitoring that complements traditional intrusion detection systems by analyzing outgoing network traffic and by identifying rogue programs in execution. A reduction in the intrusion detection search space contributes increased accuracy to the model.
- Repositories of automated attack tools on the Internet are a valuable security resource that can be analyzed and organized as to produce signatures that characterize the behavior of these programs and to detect them at runtime.
- The use of multiple software agents in the generation and matching of signatures increases detection accuracy by enabling evidence correlation from various data sources and creates a protection mechanism capable of graceful monitoring degradation in the event of system failure or compromise attempt.
- Remote attack programs in execution can be successfully identified using two basic operational aspects: the network traffic they generate, and the characteristics of process that executes them. Separate signatures can be automatically derived and a correlation method can integrate them in order to generate a more reliable matching score.

Sophos Ex. 1007
Page 153 of 216

- Attack signatures supported by an ongology can be easily deployed to intrusion detection agents connected to the Internet as soon as updates become available.

- The use of an ontological model not only allows a multiagent intrusion detection system to have a clean and organized way of handling its knowledge base and its security repositories but it also aids the design and implementation phases of the system. It reduces maintenance time by guiding the definition of the dataflow and even the internal structure of agents.

- The use of virtual machines for the implementation of trusted sub-environments helps reduce the impact a malicious host has on agents by simplifying the process of verifying the integrity of the receiving environment.

- Entropy and frequency analysis of the type of data that is to be handled by an intrusion detection system allows for faster matching algorithms by exploiting the splitting power of the attributes and the locality of reference of the signatures which help speed up the classification of the input.

- The different execution variants that remote automated attack tools may have do not have significant similarities between them as to derive a single fingerprint or signature per attack program. A better approach to follow is the generation of signatures that identify each variant separately and accurately.

- The use of agent and semantic web technologies in the development of security software offers development speed advantages but, due to the functionality that comes with these tools and which is necessary to perform even the most basic functions, it imposes significant overhead to the system that results in poor overall performance of the security system and the executing platform.

- Increased detection accuracy effectively enables the development of automated response methods that allow for various types of reaction either when the input event is classified as illicit, or simply when it is abnormal at a certain degree (determined by a signature matching scoring system).

- The deployment of an agent network in charge of tracing subsequent connections of an intruder across systems connected to the Internet is a feasible way to identify the origin of an interactive connection whose main challenge is the protection of the agents against malicious hosts.

7.2 Contributions

As for contributions, we can say that this dissertation:

- Introduced the notion of outbound intrusion detection as an unselfish, cooperative way of approaching intrusion detection and discussed its advantages and disadvantages.
- Proved that outbound traffic must be inspected for signs of intrusion as much as inbound traffic is in order to detect modern security threats and attack programs.
- Improved the performance of a signature matching algorithm by using experimental data to rank signature attributes by the amount of information they convey.
- Showed how repositories of attack tools can be used as a valuable resource of security information that can be used to improve the design and expand the scope of intrusion detection software.
- Described the behavior of public-domain automated attack tools for the purpose of signature generation and detection.
- Showed that the use of multiple agents for modeling a rogue program contributes increased detection accuracy and robustness to the security mechanism.
- Proposed an intrusion detection architecture based on agent cells organized non-hierarchically and suggested the use of software sub-environments as a viable way to guaranteeing safe execution to agents.
- Proposed and implemented a source-centric ontology for an outbound intrusion detection multiagent architecture that extends ontology work in the area of intrusion detection systems.
- Showed that automated signature generation and deployment over the Internet is possible for an outbound intrusion detection system.
- Implemented a proof-of-concept prototype system for outbound intrusion detection using semantic web and agent technology exposing the advantages and disadvantages of utilizing these tools.
- Explored the problem of session tracing, proposed an agent-based solution, and described its implementation and results. borrowed from bioinformatics.

7.3 Future directions

The following are some ideas to do research derived from the results of this dissertation:

- To design OID models in which monitoring is performed within societies of trusted hosts with a well-defined membership structure. The flow of network traffic could be restricted to members of a group or society that implement OID to protect one another. This would allow us to study the benefits of security-related group memberships as well as the different incident response methods that can be used when defining groups of computers that trust one another and monitor outbound activity among them.

- To develop mechanisms that allow agents to automatically verify the integrity of the receiving sub-environment (e.g., virtual machines and interpreters) in order to minimize the risks posed by malicious hosts. This represents a more feasible way to go about host validation than trying to determine whether the system as a whole is trustworthy.

- To experiment with longer and shorter signatures in order to identify possible performance and precision gains considering variable-size and variable-attribute signatures. The fixed signature structure used in our implementation leaves room for improvement especially if signatures can be adapted to the runtime features of the programs. This could be done by selecting a smaller number of high-value attributes per program variant rather than using a set of attributes common to all programs to characterize them.

- To analyze the potential of ontological models for audit trail reduction. In order to optimize the performance of intrusion detection tools, audit trails must have a regular structure, must contain security-relevant data, and must be as compact as possible. The use of an ontology could help develop a standard for reduced audits that relied on pre-established relationships among attributes in order to infer data. That way, a shorter input would be necessary and information could be derived at the time is needed rather than having it stored without being used.

- To develop an outbound intrusion detection system utilizing supervised-learning techniques that eliminate the need for program signatures. Our research showed the dissimilarity of attack programs while in execution. This aspect can be further explored in order to determine whether a classifier is able to generalize this type of input and to detect unseen attack programs.

- To build and analyze a larger signature repository in order to identify attack-tool groups or clusters. The definition of attack-class signatures could reduce the number of total signatures the system needs to check at detection time hereby improving its responsiveness. Data mining

and machine learning techniques could be used to extract patterns that group programs together, and to detect such patterns in real time.

- To extend the system to other protocols such as UDP and to create more specific, sub-protocol signatures (e.g., ICMP signatures). The first step has been to generate per-program-variant signatures. This only contemplated a small group of TCP/IP-based attack tools but the approach could be extended to UDP and other protocols in order to discover operational differences that help design better signature layouts and algorithms.

- To develop sound, automatic ways of protecting intrusion detection evidence stored on the knowledge bases of software agents. This could be achieved through the use of encryption in such a way as to be able to use this information before a jury in security-related cases. The protection of agent data is an important challenge since agents are vulnerable to misuse by the hosting system. Methods for safeguarding agent-collected evidence in a legally-valid format are of paramount importance to the success of distributed intrusion detection.

- To integrate intrusion and outbound intrusion detection systems so that they are able to share security information and interact with each other (e.g., letting an intrusion detection tool trigger the outbound intrusion detection system when a change in the local security context appears). A more comprehensive security model might appear by interleaving the focus on local resource protection intrusion detection systems have, with the collective monitoring approach proposed by outbound intrusion detection.

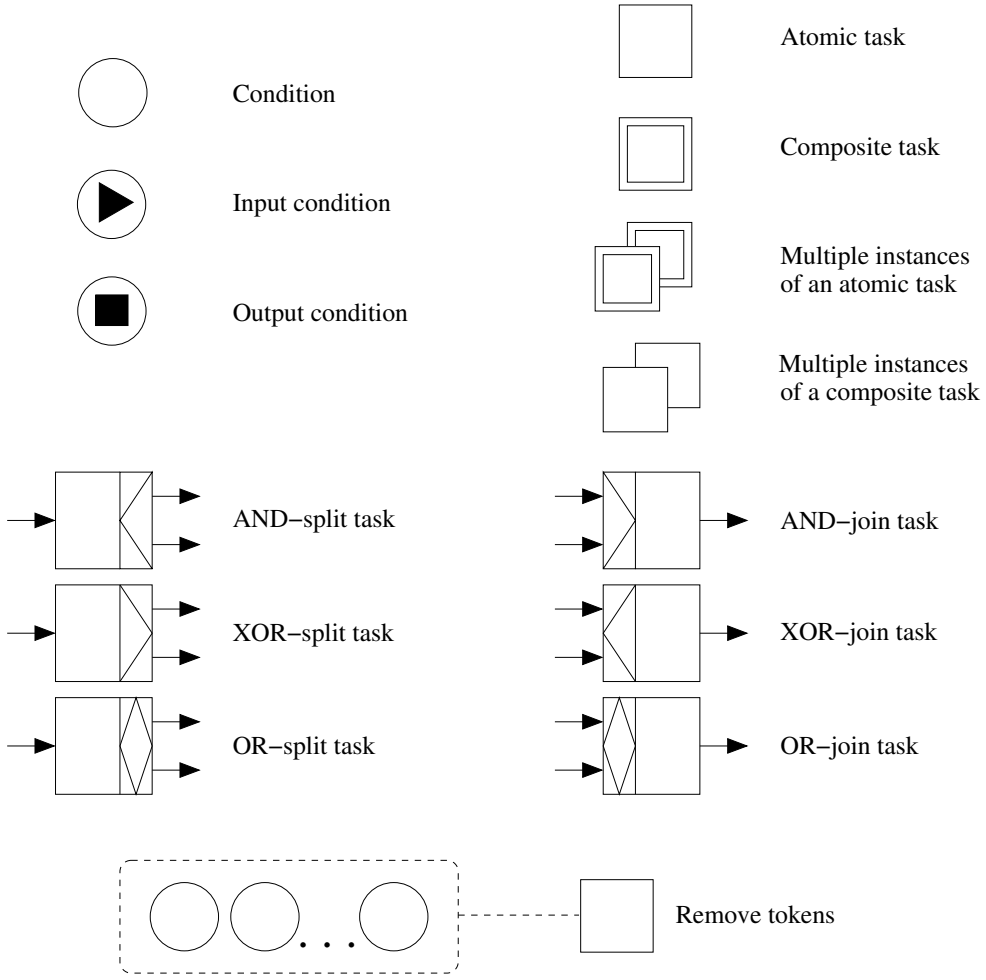
- To test the outbound intrusion detection system under extreme, real workload conditions using a very large number of attack signatures. This could help discover new ways of organizing and assigning tasks to agents, and to evaluate the functioning and design of the ontological model.

- To extend the functionality of connection-tracing agents to the exchange of encrypted session information among trusted hosts so that suspicious activity can also be detected when encrypted channels are used to compromise a system. Currently, network-based intrusion detection systems are blind to encrypted traffic. If agents placed at different hosts are able to share the contents of encrypted sessions originated at their sites, it could be possible to discover (either, with the traditional intrusion detection approach, or with outbound intrusion detection) an attack in progress that is using ciphered channels.

- To test the FROID system with distributed denial of service tools (DDoS) so that signatures for this type of attack can be generated and deployed as soon as a copy of the tool becomes available. This could help shed some light on the automatic generation of execution signatures for worms and not only for humanly-operated programs.

Appendix A: YAWL symbols

YAWL (*Yet Another Workflow Language*) is a very powerful workflow language based on a rigorous survey of the best of breed solutions on the market [206]. Its constructs are inspired by an advanced concurrency control language, Petri Nets. The basic symbols used to create workflow diagrams in YAWL are the following:



Sophos Ex. 1007
Page 159 of 216

Appendix B: TCP and IP datagram formats

B.1 TCP datagram format

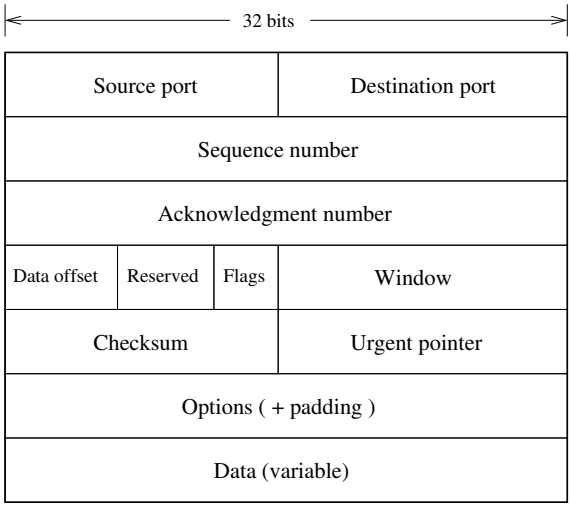


Figure 7.1: TCP datagram format (RFC 793) [38].

Together with Internet Protocol (IP), TCP is one of the core protocols underlying the Internet. They are usually referred to as a group by the term "TCP/IP". TCP enables two computers to establish a connection and exchange information. TCP guarantees delivery of data and also guarantees that information packets will be delivered in the same order in which they were sent. The fields of the TCP packet are described here [38]:

- **Source port and destination port**. Identify the points (sockets) at which upper-layer source and destination processes receive TCP services.
- **Sequence number**. Usually specifies the number assigned to the first byte of data in the current message. Under certain circumstances, it can also be used to identify an initial sequence number to be used in the upcoming transmission.

Sophos Ex. 1007
Page 161 of 216

- **Acknowledgment number**. Contains the sequence number of the next byte of data that the sender of the packet expects to receive.
- **Data offset**. Indicates the number of 32-bit words in the TCP header.
- **Reserved**. Is reserved for future use.
- **Flags**. Carries a variety of control information.
- **Window**. Specifies the size of the senderś receive window (buffer space available for incoming data).
- **Checksum**. Provides information used to determine whether the header was damaged in transit.
- **Urgent pointer**. Points to the first urgent data byte in the packet.
- **Options**. Specifies various TCP options.
- **Data**. Contains upper-layer information.

Example

TCP HEADER FORMAT

```
OCTET 1,2 Source Port (SRC_PORT)
OCTET 3,4 Destination Port (DEST_PORT)
OCTET 5,6,7,8 Sequence Number (SEQ)
OCTET 9,10,11,12 Acknowledgement Number (ACK)
OCTET 13,14 Data Offset (4 bit)+Reserved (6 bit)+
Control Flags(6 bit) (DTO, FLG)
OCTET 15,16 Window (WIN)
OCTET 17,18 Checksum (TCP_SUM)
OCTET 19,20  Urgent Pointer (URP)
OCTET 21,22,23  Options (OPT)
OCTET 24 Padding
OCTET 25,26... Data
```

TCP/IP Packet Example:

```
7E 21 45 00 00 4B 57 49 40 00 FA 06 85 77 C7 B6 78 0E CE D6 95 50 00 6E 04 9F 74 5B EE A2
59 9A 00 0E 50 18 24 00 E3 2A 00 00 2B 4F 4B 20 50 61 73 73 77 6F 72 64 20 72 65 71 75 69
72 65 64 20 66 6F 72 20 61 6C 65 78 75 72 2E 0D 0A 67 B2 7E

Start 7E
SEP 21
IP Header 45 00 00 4B 57 49 40 00 FA 06 85 77 C7 B6 78 0E CE D6 95 50
TCP Header 00 6E 04 9F 74 5B EE A2 59 9A 00 0E 50 18 24 00 E3 2A 00 00
Data 2B 4F 4B 20 50 61 73 73 77 6F 72 64 20 72 65 71 75 69 72 65 64 20 66 6F 72
20 61 6C 65 78 75 72 2E 0D 0A
FCS 67 B2
Stop 7E
```

TCP Header:

```
SRC_PORT=110  DEST_PORT=1183   SEQ=745BEEA2  ACK=599A000E   DTO=5  FLG=18  WIND=9216
TCP_SUM=E32A  URP=0000   (No Options)
```

Data:

```
+OK Password required for alexur\r\n

Control Flags (FLG=18):

FLG=00011000 Urgent Pointer  URG=0
Acknowledgment ACK=1
Push Function PSH=1
Reset connection RST=0
Synchronization SYN=0
Finished data FIN=0

State = ACK-PSH
```

B.2 IPv4 datagram format

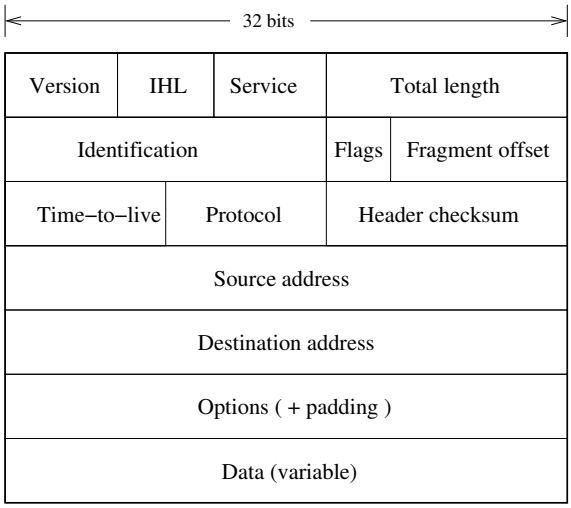


Figure 7.2: IP v.4 datagram format (RFC 791) [38].

The Internet Protocol (IP) provides a frame for encapsulating other protocols like TCP and UDP. The IP header informs the recipient among other things of: the destination and source addresses of the packet, number of octets in the packet, whether the packet can be fragmented or not, how many hops can the packet traverse, the protocol that the packet carries, etc. The IP version currently utilizied is 4. The fields of the IP packet are as follows [38]:

- **Version**. Indicates the version of this IP datagram.

- **IP Header Length (IHL)**. Indicates the datagram header length in 32-bit words.
- **Service or Type of Service**. Specifies how a particular upper-layer protocol would like the current datagram to be handled. Datagrams can be assigned various levels of importance using this field.

  Today this field is used primarily to provide quality of service (QoS) capabilities to TCP/IP for applications requiring predictable bandwidth or delay. RFC 2474 describes a method by which the TOS field is replaced by a DS field that is used to provide differentiated services (DiffServ) on networks. This field is split into two parts. The first 6 bits are used for the DSCP codepoint, which is used to differentiate traffic. The last 2 bits, or CU, are ignored by DiffServ-compliant nodes.
- **Total Length**. Specifies the length of the entire IP packet, including data and header, in bytes.
- **Identification**. Consists of an integer identifying this datagram. This field is used to help piece together datagram fragments.
- **Flags**. Consists of 3 bits, of which the low-order 2 bits control fragmentation. One bit specifies whether the packet can be fragmented; the second bit specifies whether the packet is the last fragment in a series of fragmented packets.
- **Time-to-Live**. Maintains a counter that gradually decrements down to zero, at which point the datagram is discarded. This keeps packets from looping endlessly.
- **Protocol**. Indicates which upper-layer protocol receives incoming packets after IP processing is complete.
- **Header Checksum**. Helps ensure IP header integrity.
- **Source Address**. Specifies the sending node.
- **Destination Address**. Specifies the receiving node.
- **Options**. Allows IP to support various options, such as security.
- **Data**. Contains upper-layer information.

Example

IP HEADER FORMAT

```
OCTET 1 Version (4 bit)+IHL (4 bit) (VER, IHL)
OCTET 2 Type of service (TOS)
OCTET 3,4 Total Length (TOL)
OCTET 5,6  Identification (ID)
OCTET 7,8 Flags (3 bit)+Fragment Offset (13 bit) (FLG, FRO)
OCTET 9 Time to Live (TTL)
OCTET 10 Protocol (PRO)
OCTET 11,12  Header Checksum (IP_SUM)
```

```
OCTET 13,14,15,16  Source Address (SRC)
OCTET 17,18,19,20 Destination Address (DEST)
OCTET 21,22,23 Options (OPT)
OCTET 24 Padding
OCTET 25, 26 ... Data
```

In the example shown the IP packet is encapsulated in a PPP frame including the flag sequence (7E), address (FF 03), protocol (00 21) and FCS (0B 81). In turn, the IP packet encapsulates a UDP message.

Example:

```
7E FF 03 00 21 45 00 00 40 00 01 00 00 3C 11 E0 31 CE D9 8F 1F C7 B6 78 CB 04 63 00
35 00 2C AB DA 00 01 01 00 00 01 00 00 00 00 00 00 04 70 6F 70 64 02 69 78 06 6E 65
74 63 6F 6D 03 63 6F 6D 00 00 01 00 01 0B 81 7E
```

```
Start 7E
Address FF 03
SEP 00 21
IP Header 45 00 00 40 00 01 00 00 3C 11 E0 31 CE D9 8F 1F C7 B6 78 CB
Data 04 89 00 35 00 2C AB B4 00 01 01 00 00 01 00 00 00 00 00 00 04 70 6F
70 64 02 69 78 06 6E 65 74 63 6F 6D 03 63 6F 6D 00 00 01 00 01
FCS 0B 81
Stop 7E
```

IP Header:

```
VER=4 IHL=5 TOS=0 TOL=64 ID=1 FLG=00 FRO=00 TTL=60 PRO=17 IP_SUM=E031
SRC=206.217.143.31. DEST=199.182.120.203. OPT=00000000
```

Appendix C: Outbound intrusion detection ontology in OWL

This is the top level definition of the source-centric OID ontology that supports the proposed architecture (Figure 4.12). It has been created in OWL [7] using the Protégé ontology definition tool [144].

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://localhost/OIDontology#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://localhost/OIDontology">
  <owl:Ontology rdf:about="OIDontology">
    <owl:imports rdf:resource="http://protege.stanford.edu/plugins/owl/protege"/>
  </owl:Ontology>
  <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
      <rdf:Description rdf:about="http://www.w3.org/2002/07/owl#Thing"/>
      <owl:Class rdf:about="#TrafficSignature"/>
    </owl:unionOf>
  </owl:Class>
  <owl:Class rdf:ID="Tracer">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Reactor"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="ProcessMonitoringCell">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Cell"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="IPpacket"/>
  <owl:Class rdf:ID="ProcessSignature">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Signature"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="CompiledExecutable">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Executable"/>
```


Sophos Ex. 1007
Page 167 of 216

```
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="ProcessSigner">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Signer"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="TrafficSigner">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Signer"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="HostileProcess">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Process"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Reaction"/>
<owl:Class rdf:ID="ProcessSensor">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Sensor"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="EvidenceRepository">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Repository"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="TrafficDetector">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Detector"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="ActiveReaction">
  <rdfs:subClassOf rdf:resource="#Reaction"/>
</owl:Class>
<owl:Class rdf:ID="ProcessSigningCell">
  <rdfs:subClassOf rdf:resource="#Cell"/>
</owl:Class>
<owl:Class rdf:ID="TrafficSensor">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Sensor"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Reactor">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Agent"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="ProcessDetector">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Detector"/>
  </rdfs:subClassOf>
</owl:Class>
```

```
<owl:Class rdf:ID="SignatureRepository">
  <rdfs:subClassOf rdf:resource="#Repository"/>
</owl:Class>
<owl:Class rdf:ID="SafeProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
</owl:Class>
<owl:Class rdf:ID="TCPpacket">
  <rdfs:subClassOf rdf:resource="#IPpacket"/>
</owl:Class>
<owl:Class rdf:ID="DataSource"/>
<owl:Class rdf:ID="InterpretedExecutable">
  <rdfs:subClassOf rdf:resource="#Executable"/>
</owl:Class>
<owl:Class rdf:ID="InternalData">
  <rdfs:subClassOf rdf:resource="#DataSource"/>
</owl:Class>
<owl:Class rdf:ID="TrafficSignature">
  <rdfs:subClassOf rdf:resource="#Signature"/>
</owl:Class>
<owl:Class rdf:ID="Correlator">
  <rdfs:subClassOf rdf:resource="#Agent"/>
</owl:Class>
<owl:Class rdf:ID="HybridDetector">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Detector"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Signer">
  <rdfs:subClassOf rdf:resource="#Correlator"/>
</owl:Class>
<owl:Class rdf:ID="TrafficMonitoringCell">
  <rdfs:subClassOf rdf:resource="#Cell"/>
</owl:Class>
<owl:Class rdf:ID="UDPpacket">
  <rdfs:subClassOf rdf:resource="#IPpacket"/>
</owl:Class>
<owl:Class rdf:ID="Sensor">
  <rdfs:subClassOf rdf:resource="#Agent"/>
</owl:Class>
<owl:Class rdf:ID="Guard">
  <rdfs:subClassOf rdf:resource="#Reactor"/>
</owl:Class>
<owl:Class rdf:ID="Detector">
  <rdfs:subClassOf rdf:resource="#Correlator"/>
</owl:Class>
<owl:Class rdf:ID="CorrelationCell">
  <rdfs:subClassOf rdf:resource="#Cell"/>
</owl:Class>
<owl:Class rdf:ID="OutgoingData">
  <rdfs:subClassOf rdf:resource="#DataSource"/>
</owl:Class>
<owl:Class rdf:ID="TrafficSigningCell">
  <rdfs:subClassOf rdf:resource="#Cell"/>
</owl:Class>
```

```
<owl:Class rdf:ID="PassiveReaction">
  <rdfs:subClassOf rdf:resource="#Reaction"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="container">
  <rdfs:domain rdf:resource="#Signature"/>
  <rdfs:range rdf:resource="#SignatureRepository"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="member">
  <rdfs:domain rdf:resource="#Agent"/>
  <rdfs:range rdf:resource="#Cell"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="trafficSource">
  <rdfs:domain rdf:resource="#TrafficSensor"/>
  <rdfs:range rdf:resource="#OutgoingData"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="trafficSupplier">
  <rdfs:domain rdf:resource="#HybridDetector"/>
  <rdfs:range rdf:resource="#TrafficDetector"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="executant">
  <rdfs:range rdf:resource="#Reactor"/>
  <rdfs:domain rdf:resource="#Correlator"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="processSource">
  <rdfs:range rdf:resource="#InternalData"/>
  <rdfs:domain rdf:resource="#ProcessSensor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="effect">
  <rdfs:range rdf:resource="#Reaction"/>
  <rdfs:domain rdf:resource="#Reactor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="evidence">
  <rdfs:range rdf:resource="#Executable"/>
  <rdfs:domain rdf:resource="#PassiveReaction"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="item">
  <rdfs:range rdf:resource="#IPpacket"/>
  <rdfs:domain rdf:resource="#OutgoingData"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="processSupplier">
  <rdfs:domain rdf:resource="#HybridDetector"/>
  <rdfs:range rdf:resource="#ProcessDetector"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="cause">
  <rdfs:range rdf:resource="#HostileProcess"/>
  <rdfs:domain rdf:resource="#PassiveReaction"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="processProduct">
  <rdfs:domain rdf:resource="#ProcessSigner"/>
  <rdfs:range rdf:resource="#ProcessSignature"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="storage">
  <rdfs:range rdf:resource="#InternalData"/>
  <rdfs:domain rdf:resource="#Process"/>
```

```
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="execution">
    <rdfs:range rdf:resource="#Process"/>
    <rdfs:domain rdf:resource="#Executable"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="proof">
    <rdfs:domain rdf:resource="#EvidenceRepository"/>
    <rdfs:range rdf:resource="#Executable"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="informant">
    <rdfs:range rdf:resource="#Sensor"/>
    <rdfs:domain rdf:resource="#Correlator"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="trafficProduct">
    <rdfs:range rdf:resource="#TrafficSignature"/>
    <rdfs:domain rdf:resource="#TrafficSigner"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="description">
    <rdfs:range rdf:resource="#EvidenceRepository"/>
    <rdfs:domain rdf:resource="#Agent"/>
  </owl:ObjectProperty>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 1.2 beta, Build 139)  http://protege.stanford.edu -->
```

Appendix D: Classification Trees

These trees were generated using Gnome Data Miner's decision trees module [69].

Entropy tree

```
Splitting function: Entropy

/*-----------------------------------------------------------------
  domains
-----------------------------------------------------------------*/
dom(packetCounter) = ZZ [2, 16];
dom(fragmentation) = { 0, 1 };
dom(sourceAddress) = IR [-8205, -92];
dom(destinationAddress) = ZZ [-9720, -2322];
dom(destinationPort1) = ZZ [1, 8080];
dom(destinationport2) = ZZ [0, 65535];
dom(windowSize) = { 0, 1 };
dom(h8) = ZZ [0, 3360];
dom(h4) = ZZ [0, 3256];
dom(h5) = ZZ [0, 3360];
dom(f3) = ZZ [0, 8];
dom(f4) = ZZ [0, 9];
dom(f7) = ZZ [0, 8];
dom(f8) = ZZ [0, 8];
dom(f10) = ZZ [0, 8];
dom(f15) = ZZ [0, 8];
dom(l4) = ZZ [0, 1448];
dom(l6) = ZZ [0, 1448];
dom(programID) = { 1a, 1b, 2a, 2b, 2c, 3a, 4a, 4b, 5a, 5b, 6a, 6b,
  7a, 7b, 8a, 8b, 9a, 9b, 10a, 10b, 11a, 12a, 12b, 13a, 14a, 15a,
  16a, 17a, 18a, 18b, 19a, 20a, 21a, 22a, 23a, 24a, 25a, 26a, 27a,
  27b, 28a, 29a, 30a, 31a, 32a, 33a, 34a, 35a, 36a, 36b, 37a, 38a,
  39a, 40a, 41a, 42a, 43a, 44a, 45a, 46a, 47a, 48a, 49a, 50a };

/*-----------------------------------------------------------------
  decision tree
-----------------------------------------------------------------*/
dtree(programID) =
{ (l4|29)
  <:{ (f7|0.5)
      <:{ (packetCounter|4.5)
```


Sophos Ex. 1007
Page 173 of 216

```
        <:{ (h8|1075)
            <:{ (packetCounter|2.5)
                <:{ (destinationPort1|37)
                    <:{ 37a: 6 },
                    >:{ 34a: 5 }},
                >:{ (packetCounter|3.5)
                    <:{ 1a: 5 },
                    >:{ 36a: 5 }}},
            >:{ (destinationPort1|480)
                <:{ (sourceAddress|-4148.5)
                    <:{ 39a: 5 },
                    >:{ 29a: 5 }},
                >:{ (packetCounter|2.5)
                    <:{ 17a: 5 },
                    >:{ 31a: 5 }}}},
        >:{ (packetCounter|5.5)
            <:{ (f3|6)
                <:{ (destinationAddress|-3716)
                    <:{ 25a: 5 },
                    >:{ 11a: 5 }},
                >:{ (h5|4905)
                    <:{ 8b: 5 },
                    >:{ 4a: 5 }}},
            >:{ (packetCounter|9.5)
                <:{ (packetCounter|7)
                    <:{ 23a: 5 },
                    >:{ 24a: 5 }},
                >:{ (packetCounter|12)
                    <:{ 15a: 5 },
                    >:{ 20a: 5 }}}}},
    >:{ (h5|1035)
        <:{ (destinationPort1|24)
            <:{ (packetCounter|9)
                <:{ (destinationPort1|22.5)
                    <:{ 9a: 5 },
                    >:{ 1b: 5 }},
                >:{ (destinationport2|32767.5)
                    <:{ 27a: 5 },
                    >:{ 35a: 5 }}},
            >:{ (destinationPort1|3066)
                <:{ (packetCounter|8.5)
                    <:{ 10b: 5 },
                    >:{ 49a: 5 }},
                >:{ (packetCounter|7.5)
                    <:{ 7b: 5 },
                    >:{ 44a: 5 }}}},
        >:{ (packetCounter|14)
            <:{ (destinationPort1|51.5)
                <:{ (packetCounter|11.5)
                    <:{ 36b: 5 },
                    >:{ 14a: 5 }},
                >:{ (destinationPort1|111.5)
                    <:{ 32a: 5 },
                    >:{ 21a: 5 }}},
```

```
            >:{ (destinationPort1|348.5)
                <:{ (destinationPort1|40.5)
                    <:{ 41a: 5 },
                    >:{ 48a: 5 }},
                >:{ (destinationPort1|3320)
                    <:{ 46a: 5 },
                    >:{ 7a: 5 }}}}}},
>:{ (l4|641)
    <:{ (packetCounter|13)
        <:{ (packetCounter|7.5)
            <:{ (packetCounter|6.5)
                <:{ (packetCounter|5.5)
                    <:{ 6b: 5 },
                    >:{ 16a: 5 }},
                >:{ (fragmentation)
                    0:{ 38a: 5 },
                    1:{ 42a: 5 }}},
            >:{ (windowSize)
                0:{ (destinationPort1|51.5)
                    <:{ 47a: 5 },
                    >:{ 13a: 5 }},
                1:{ (packetCounter|10.5)
                    <:{ 43a: 5 },
                    >:{ 40a: 5 }}}},
        >:{ (f8|4)
            <:{ (windowSize)
                0:{ (packetCounter|15)
                    <:{ 28a: 5 },
                    >:{ 8a: 5 }},
                1:{ (packetCounter|15.5)
                    <:{ 45a: 5 },
                    >:{ 30a: 5 }}},
            >:{ (f15|4)
                <:{ (destinationport2|55)
                    <:{ 18b: 5 },
                    >:{ 6a: 5 }},
                >:{ (packetCounter|15)
                    <:{ 33a: 5 },
                    >:{ 18a: 5 }}}}},
    >:{ (h4|2433)
        <:{ (f10|0.5)
            <:{ (destinationPort1|51)
                <:{ (l6|724)
                    <:{ 10a: 5 },
                    >:{ 2b: 5 }},
                >:{ (packetCounter|9)
                    <:{ 4b: 5 },
                    >:{ 50a: 5 }}},
            >:{ (packetCounter|15)
                <:{ (packetCounter|12)
                    <:{ 3a: 5 },
                    >:{ 9b: 5 }},
                >:{ (f4|3)
                    <:{ 2a: 5 },
```

```
                    >:{ 2c: 5 }}}},
          >:{ (packetCounter|7.5)
              <:{ (windowSize)
                  0:{ (destinationPort1|51.5)
                      <:{ 12a: 5 },
                      >:{ 12b: 5 }},
                  1:{ (packetCounter|6.5)
                      <:{ 19a: 5 },
                      >:{ 26a: 5 }}},
              >:{ (f7|4)
                  <:{ (packetCounter|12)
                      <:{ 5b: 5 },
                      >:{ 5a: 5 }},
                  >:{ (l6|1317)
                      <:{ 27b: 5 },
                      >:{ 22a: 5 }}}}}};

/*-------------------------------------------------------------------
  number of attributes: 19
  tree height         : 7
  number of nodes     : 127
  number of tuples    : 321
-------------------------------------------------------------------*/
```

Information Gain tree

```
Splitting function: Information Gain

/*-------------------------------------------------------------------
  domains
-------------------------------------------------------------------*/
dom(packetCounter) = ZZ [2, 16];
dom(fragmentation) = { 0, 1 };
dom(sourceAddress) = IR [-8205, -92];
dom(destinationAddress) = ZZ [-9720, -2322];
dom(destinationPort1) = ZZ [1, 8080];
dom(destinationport2) = ZZ [0, 65535];
dom(windowSize) = { 0, 1 };
dom(stdDev) = { 0, 30 };
dom(h5) = ZZ [0, 3360];
dom(h8) = ZZ [0, 3360];
dom(h4) = ZZ [0, 3256];
dom(f4) = ZZ [0, 9];
dom(f7) = ZZ [0, 8];
dom(f8) = ZZ [0, 8];
dom(f15) = ZZ [0, 8];
dom(l6) = ZZ [0, 1448];
dom(programID) = { 1a, 1b, 2a, 2b, 2c, 3a, 4a, 4b, 5a, 5b, 6a, 6b,
  7a, 7b, 8a, 8b, 9a, 9b, 10a, 10b, 11a, 12a, 12b, 13a, 14a, 15a,
  16a, 17a, 18a, 18b, 19a, 20a, 21a, 22a, 23a, 24a, 25a, 26a, 27a,
  27b, 28a, 29a, 30a, 31a, 32a, 33a, 34a, 35a, 36a, 36b, 37a, 38a,
```

```
  39a, 40a, 41a, 42a, 43a, 44a, 45a, 46a, 47a, 48a, 49a, 50a };

/*-------------------------------------------------------------------
  decision tree
-------------------------------------------------------------------*/
dtree(programID) =
{ (packetCounter|2.5)
  <:{ (destinationPort1|37)
      <:{ 37a: 6 },
      >:{ (sourceAddress|-4148.5)
          <:{ (destinationPort1|1473.5)
              <:{ (h8|934)
                  <:{ 34a: 5 },
                  >:{ 39a: 5 }},
              >:{ 17a: 5 }},
          >:{ 29a: 5 }}},
  >:{ (stdDev)
      0 :{ (windowSize)
           0:{ (fragmentation)
               0:{ (packetCounter|15)
                   <:{ (packetCounter|4)
                       <:{ 1a: 5 },
                       >:{ (h5|1292)
                           <:{ (packetCounter|10.5)
                               <:{ (packetCounter|5.5)
                                   <:{ (destinationAddress|-3716)
                                       <:{ (destinationPort1|3042)
                                           <:{ (destinationPort1|51)
                                               <:{ (destinationPort1|20.5)
                                                   <:{ 8b: 5 },
                                                   >:{ 6b: 5 }},
                                               >:{ 4a: 5 }},
                                           >:{ 25a: 5 }},
                                       >:{ 11a: 5 }},
                                   >:{ (packetCounter|6.5)
                                       <:{ (destinationAddress|-7415)
                                           <:{ 16a: 5 },
                                           >:{ 23a: 5 }},
                                       >:{ (h8|2325)
                                           <:{ (packetCounter|8.5)
                                               <:{ (packetCounter|7.5)
                                                   <:{ (destinationport2|32767.5)
                                                       <:{ 12b: 5 },
                                                       >:{ 38a: 5 }},
                                                   >:{ (destinationPort1|7080)
                                                       <:{ (destinationPort1|3095)
                                                           <:{ (destinationPort1|22.5)
                                                               <:{ 9a: 5 },
                                                               >:{ (destinationPort1|95)
                                                                   <:{ (h8|5135)
                                                                       <:{ 47a: 5 },
                                                                       >:{ (h5|2232)
                                                                           <:{ 13a: 5 },
                                                                           >:{ 4b: 5 }
```

```
                                                                    }
                                                                 },
                                                              >:{ 10b: 5 }
                                                           }},
                                                        >:{ 44a: 5 }
                                                     },
                                                  >:{ 24a: 5 }}},
                                        >:{ (destinationPort1|23.5)
                                            <:{ (l6|724)
                                                <:{ 10a: 5 },
                                                >:{ 2b: 5 }},
                                            >:{ (destinationPort1|52.5)
                                                <:{ 49a: 5 },
                                                >:{ (l6|1317)
                                                    <:{ 27b: 5 },
                                                    >:{ 22a: 5 }
                                                }}}},
                                    >:{ (destinationPort1|51)
                                        <:{ (destinationport2|32767.5)
                                            <:{ 27a: 5 },
                                            >:{ 35a: 5 }},
                                        >:{ (destinationport2|32767.5)
                                            <:{ 50a: 5 },
                                            >:{ 3a: 5 }}}}}},
                            >:{ (destinationPort1|126.5)
                                <:{ (destinationPort1|95)
                                    <:{ (destinationPort1|51)
                                        <:{ 36b: 5 },
                                        >:{ 32a: 5 }},
                                    >:{ 15a: 5 }},
                                >:{ 21a: 5 }}},
                        >:{ (destinationPort1|50.5)
                            <:{ 28a: 5 },
                            >:{ (destinationport2|55)
                                <:{ 33a: 5 },
                                >:{ 9b: 5 }}}}},
>:{ (h8|1135)
    <:{ 41a: 5 },
    >:{ (destinationPort1|348.5)
        <:{ (destinationPort1|51)
            <:{ 5a: 5 },
            >:{ (destinationport2|66)
                <:{ (destinationport2|11)
                    <:{ (f4|3)
                        <:{ (f7|4.5)
                            <:{ (f8|4)
                                <:{ 2a: 5 },
                                >:{ (f15|4)
                                    <:{ 18b: 5 },
                                    >:{ 18a: 5 }}},
                            >:{ 8a: 5 }},
                        >:{ 2c: 5 }},
                    >:{ 48a: 5 }},
                <:{ 6a: 5 }}},
```

```
                        >:{ 46a: 5 }}}},
                1:{ (packetCounter|7.5)
                    <:{ (packetCounter|5.5)
                        <:{ 36a: 5 },
                        >:{ 42a: 5 }},
                    >:{ 5b: 5 }}},
            1:{ (packetCounter|15.5)
                <:{ (packetCounter|14)
                    <:{ (packetCounter|11)
                        <:{ (packetCounter|4.5)
                            <:{ 31a: 5 },
                            >:{ (packetCounter|8)
                                <:{ (packetCounter|6.5)
                                    <:{ 19a: 5 },
                                    >:{ (destinationPort1|3022)
                                        <:{ 26a: 5 },
                                        >:{ 7b: 5 }}},
                                >:{ 43a: 5 }}},
                        >:{ 20a: 5 }},
                    >:{ 45a: 5 }},
                >:{ (destinationPort1|3022)
                    <:{ 30a: 5 },
                    >:{ 7a: 5 }}}},
      30:{ (packetCounter|10)
            <:{ (packetCounter|7.5)
                <:{ 12a: 5 },
                >:{ 1b: 5 }},
            >:{ (windowSize)
                0:{ 14a: 5 },
                1:{ 40a: 5 }}}}};

/*-------------------------------------------------------------------
  number of attributes: 17
  tree height         : 20
  number of nodes     : 127
  number of tuples    : 321
-------------------------------------------------------------------*/
```

GINI tree

```
Splitting function: GINI

/*-------------------------------------------------------------------
  domains
-------------------------------------------------------------------*/
dom(packetCounter) = ZZ [2, 16];
dom(fragmentation) = { 0, 1 };
dom(sourceAddress) = IR [-8205, -92];
dom(destinationAddress) = ZZ [-9720, -2322];
dom(destinationPort1) = ZZ [1, 8080];
dom(destinationport2) = ZZ [0, 65535];
dom(windowSize) = { 0, 1 };
```

```
dom(h4) = ZZ [0, 3256];
dom(h5) = ZZ [0, 3360];
dom(h8) = ZZ [0, 3360];
dom(f4) = ZZ [0, 9];
dom(f7) = ZZ [0, 8];
dom(f8) = ZZ [0, 8];
dom(f15) = ZZ [0, 8];
dom(l6) = ZZ [0, 1448];
dom(programID) = { 1a, 1b, 2a, 2b, 2c, 3a, 4a, 4b, 5a, 5b, 6a, 6b,
  7a, 7b, 8a, 8b, 9a, 9b, 10a, 10b, 11a, 12a, 12b, 13a, 14a, 15a,
  16a, 17a, 18a, 18b, 19a, 20a, 21a, 22a, 23a, 24a, 25a, 26a, 27a,
  27b, 28a, 29a, 30a, 31a, 32a, 33a, 34a, 35a, 36a, 36b, 37a, 38a,
  39a, 40a, 41a, 42a, 43a, 44a, 45a, 46a, 47a, 48a, 49a, 50a };

/*-------------------------------------------------------------------
  decision tree
-------------------------------------------------------------------*/
dtree(programID) =
{ (packetCounter|15.5)
  <:{ (windowSize)
      0:{ (packetCounter|13)
          <:{ (packetCounter|11.5)
              <:{ (f7|4)
                  <:{ (h4|2338)
                      <:{ (packetCounter|6.5)
                          <:{ (packetCounter|5.5)
                              <:{ (packetCounter|2.5)
                                  <:{ (sourceAddress|-4148.5)
                                      <:{ (destinationPort1|1473.5)
                                          <:{ (destinationPort1|94)
                                              <:{ (destinationPort1|37)
                                                  <:{ 37a: 6 },
                                                  >:{ 34a: 5 }},
                                              >:{ 39a: 5 }},
                                          >:{ 17a: 5 }},
                                      >:{ 29a: 5 }},
                                  >:{ (packetCounter|4.5)
                                      <:{ (packetCounter|3.5)
                                          <:{ 1a: 5 },
                                          >:{ 36a: 5 }},
                                      >:{ (destinationAddress|-3716)
                                          <:{ (destinationPort1|3042)
                                              <:{ (destinationPort1|51)
                                                  <:{ (destinationPort1|20.5)
                                                      <:{ 8b: 5 },
                                                      >:{ 6b: 5 }},
                                                  >:{ 4a: 5 }},
                                              >:{ 25a: 5 }},
                                          >:{ 11a: 5 }}}},
                              >:{ (destinationAddress|-7415)
                                  <:{ 16a: 5 },
                                  >:{ 23a: 5 }}},
                          >:{ (packetCounter|7.5)
                              <:{ (fragmentation)
```

```
                        0:{ (destinationPort1|51.5)
                            <:{ 12a: 5 },
                            >:{ (destinationport2|32767.5)
                                <:{ 12b: 5 },
                                >:{ 38a: 5 }}},
                        1:{ 42a: 5 }},
                    >:{ (fragmentation)
                        0:{ (destinationPort1|7080)
                            <:{ (destinationPort1|3095)
                                <:{ (destinationPort1|95)
                                    <:{ (destinationPort1|51.5)
                                        <:{ (destinationPort1|22.5)
                                            <:{ 9a: 5 },
                                            >:{ (destinationport2|4595.5)
                                                <:{ 47a: 5 },
                                                >:{ 1b: 5 }}
                                          },
                                        >:{ (destinationport2|32767.5)
                                            <:{ 13a: 5 },
                                            >:{ 4b: 5 }}},
                                    >:{ 10b: 5 }},
                                >:{ 44a: 5 }},
                            >:{ 24a: 5 }},
                        1:{ 5b: 5 }}}},
                >:{ (packetCounter|9.5)
                    <:{ (destinationPort1|52.5)
                        <:{ (destinationPort1|23.5)
                            <:{ (16|724)
                                <:{ 10a: 5 },
                                >:{ 2b: 5 }},
                            >:{ 49a: 5 }},
                        >:{ (16|1317)
                            <:{ 27b: 5 },
                            >:{ 22a: 5 }}},
                    >:{ (destinationPort1|51)
                        <:{ (destinationport2|32767.5)
                            <:{ 27a: 5 },
                            >:{ 35a: 5 }},
                        >:{ (destinationport2|32767.5)
                            <:{ 50a: 5 },
                            >:{ 3a: 5 }}}}},
            >:{ (destinationPort1|126.5)
                <:{ (destinationPort1|95)
                    <:{ (destinationPort1|51)
                        <:{ 36b: 5 },
                        >:{ 32a: 5 }},
                    >:{ 15a: 5 }},
                >:{ 21a: 5 }}},
        >:{ 14a: 5 }},
    >:{ (destinationPort1|50.5)
        <:{ 28a: 5 },
        >:{ (destinationport2|55)
            <:{ 33a: 5 },
            >:{ 9b: 5 }}}},
```

```
      1:{ (packetCounter|14)
          <:{ (packetCounter|12.5)
              <:{ (packetCounter|10.5)
                  <:{ (packetCounter|8)
                      <:{ (packetCounter|6.5)
                          <:{ (packetCounter|4.5)
                              <:{ 31a: 5 },
                              >:{ 19a: 5 }},
                          >:{ (destinationPort1|3022)
                              <:{ 26a: 5 },
                              >:{ 7b: 5 }}},
                      >:{ 43a: 5 }},
                  >:{ 40a: 5 }},
              >:{ 20a: 5 }},
          >:{ 45a: 5 }}},
  >:{ (destinationPort1|3320)
      <:{ (f4|3)
          <:{ (h5|52341)
              <:{ (destinationPort1|21.5)
                  <:{ (destinationPort1|11)
                      <:{ 41a: 5 },
                      >:{ 30a: 5 }},
                  >:{ 5a: 5 }},
              >:{ (destinationport2|66)
                  <:{ (h8|321)
                      <:{ (f4|3)
                          <:{ (f7|4.5)
                              <:{ (f8|4)
                                  <:{ 2a: 5 },
                                  >:{ (f15|4)
                                      <:{ 18b: 5 },
                                      >:{ 18a: 5 }}},
                              >:{ 8a: 5 }},
                          >:{ 2c: 5 }},
                      >:{ 48a: 5 }},
                  >:{ 6a: 5 }}},
          >:{ 46a: 5 }},
      >:{ 7a: 5 }}};

/*-----------------------------------------------------------------------
  number of attributes: 16
  tree height         : 16
  number of nodes     : 127
  number of tuples    : 321
-----------------------------------------------------------------------*/
```

Appendix E: Process signatures

The table expanding the following pages corresponds to the process signatures of the programs surveyed in this project.

Sophos Ex. 1007
Page 183 of 216

```
<ProcessSignature rdf:ID=305monit>
        <md5> ead28b1b7133a311dcb2c0ab8c2b5c4a </md5>
        <textSegmentSize> 4103 </textSegmentSize>
        <dataSegmentSize> 508 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 3 </numArguments>
        <size> 15819 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=akill2>
        <md5> 7c57fda886a90ab8d3201a557a79ab20 </md5>
        <textSegmentSize> 0 </textSegmentSize>
        <dataSegmentSize> 0 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 1 </numArguments>
        <size> 1796 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=anubisexp>
        <md5> 4e0aa24f43a8b13ac33e06eb162dbaea </md5>
        <textSegmentSize> 10678 </textSegmentSize>
        <dataSegmentSize> 868 </dataSegmentSize>
        <arguments> -b-l-r-w </arguments>
        <numArguments> 2-6 </numArguments>
        <size> 23806 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=apache-nosejob>
        <md5> 340ffbab2f45b95c33cd3cf082282834 </md5>
        <textSegmentSize> 9821 </textSegmentSize>
        <dataSegmentSize> 972 </dataSegmentSize>
        <arguments> -h-t-o-b-d-z-r-w-c </arguments>
        <numArguments> 1-5 </numArguments>
        <size> 22523 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=apache-scalp>
        <md5> 9d4ad46076f207c90a5438734efb5632 </md5>
        <textSegmentSize> 6391 </textSegmentSize>
        <dataSegmentSize> 620 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 2 </numArguments>
        <size> 18769 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=arksink>
        <md5> 318472fdcad36d111831d6880ee83610 </md5>
        <textSegmentSize> 3759 </textSegmentSize>
        <dataSegmentSize> 376 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 2 </numArguments>
        <size> 15450 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=atphttp>
        <md5> fb654b077f6d67ce87779b15a4663e27 </md5>
        <textSegmentSize> 4095 </textSegmentSize>
        <dataSegmentSize> 420 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 1-3 </numArguments>
        <size> 15742 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=bazooka>
        <md5> db390f1e415a0b3d1d2770c892a15420 </md5>
        <textSegmentSize> 0 </textSegmentSize>
        <dataSegmentSize> 0 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 5-6 </numArguments>
        <size> 4773 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=bysin>
        <md5> c6b4738762ecc12d41420474ef4561af </md5>
        <textSegmentSize> 3885 </textSegmentSize>
        <dataSegmentSize> 336 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 2 </numArguments>
        <size> 15493 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=cge>
        <md5> 21297199a69c4b86a57ce4170dda4482 </md5>
        <textSegmentSize> 0 </textSegmentSize>
        <dataSegmentSize> 0 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 2 </numArguments>
        <size> 18530 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=cobain>
        <md5> f0cc271a17471fcd8555602dc84764d5 </md5>
        <textSegmentSize> 0 </textSegmentSize>
        <dataSegmentSize> 0 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 1 </numArguments>
        <size> 1528 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=delefate>
        <md5> c100a9d4f27fc5f4e5160f21d68e680f </md5>
        <textSegmentSize> 4396 </textSegmentSize>
        <dataSegmentSize> 864 </dataSegmentSize>
        <arguments> null </arguments>
        <numArguments> 3 </numArguments>
        <size> 16664 </size>
</ProcessSignature>
```

```
<ProcessSignature rdf:ID=emule>
       <md5> f97ec52de0388c043e7f5547f5f4b5aa </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 3 </numArguments>
       <size> 3362 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=esignal>
       <md5> 87ef0810b9c74733065752ff2e9feb8b </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 3207 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=ethboom>
       <md5> decbb70da04e8a22d3baf6f4e7938cf7 </md5>
       <textSegmentSize> 3584 </textSegmentSize>
       <dataSegmentSize> 2560 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1-2 </numArguments>
       <size> 6656 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=exceed>
       <md5> 43f3d018d0922e363781a7982db1f359 </md5>
       <textSegmentSize> 3176 </textSegmentSize>
       <dataSegmentSize> 300 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1 </numArguments>
       <size> 14301 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=extremail2>
       <md5> a6896905340d841524db18566c94dedb </md5>
       <textSegmentSize> 2745 </textSegmentSize>
       <dataSegmentSize> 312 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1 </numArguments>
       <size> 14133 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=ezploit>
       <md5> 3113afa9e6b095b1818f188542b5d56e </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 3 </numArguments>
       <size> 3993 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=fping>
       <md5> 8f7ed0db109379a6547c21c26d1e00e5 </md5>
       <textSegmentSize> 17809 </textSegmentSize>
       <dataSegmentSize> 704 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 66152 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=freezer>
       <md5> 262490c35aaa4933cb0b5b2554456cfe </md5>
       <textSegmentSize> 2816 </textSegmentSize>
       <dataSegmentSize> 312 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 5 </numArguments>
       <size> 14215 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=fspscan>
       <md5> e248a6e47086d5a3ef4238f395aa44ca </md5>
       <textSegmentSize> 5237 </textSegmentSize>
       <dataSegmentSize> 396 </dataSegmentSize>
       <arguments> o-l-p-r-t-i </arguments>
       <numArguments> 1-5 </numArguments>
       <size> 17622 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=gtkftpd>
       <md5> 47081c74be9f704d7d317aad1af1ca1e </md5>
       <textSegmentSize> 9403 </textSegmentSize>
       <dataSegmentSize> 516 </dataSegmentSize>
       <arguments> -h-P-s-u-p-c-b-a-n-r-d </arguments>
       <numArguments> 1-11 </numArguments>
       <size> 22220 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=iawebmail>
       <md5> a87b4ffcd85a199911affa76df9d3d18 </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 6316 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=ircd>
       <md5> a12e35c8195d88e27d52e4e9328b5a6f </md5>
       <textSegmentSize> 2459 </textSegmentSize>
       <dataSegmentSize> 316 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 13786 </size>
</ProcessSignature>
```


Sophos Ex. 1007
Page 185 of 216

```
<ProcessSignature rdf:ID=isspam>
       <md5> 420d323eb9c0cee5d575e9fd6557507e </md5>
       <textSegmentSize> 3384 </textSegmentSize>
       <dataSegmentSize> 916 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 3 </numArguments>
       <size> 16093 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=lsasrv>
       <md5> fff16d7ca96e0d7c1c3a734a6c0f23ea </md5>
       <textSegmentSize> 5653 </textSegmentSize>
       <dataSegmentSize> 2980 </dataSegmentSize>
       <arguments> -t </arguments>
       <numArguments> 1-5 </numArguments>
       <size> 22290 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=namedsploit>
       <md5> 98a2ff0deedb9750d7ae12ccf7137b53 </md5>
       <textSegmentSize> 6425 </textSegmentSize>
       <dataSegmentSize> 1296 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1-3 </numArguments>
       <size> 19027 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=niprint>
       <md5> 68f0946727b02511b5f921d32cb11561 </md5>
       <textSegmentSize> 2441 </textSegmentSize>
       <dataSegmentSize> 1140 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1 </numArguments>
       <size> 14642 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=ntpd>
       <md5> 063e963f3d5cec5daf7e6936e6fd3740 </md5>
       <textSegmentSize> 4140 </textSegmentSize>
       <dataSegmentSize> 688 </dataSegmentSize>
       <arguments> -t </arguments>
       <numArguments> 2 </numArguments>
       <size> 16305 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=numby>
       <md5> 9fb7a5ebb06c48a284b9c2f1bfe3ec82 </md5>
       <textSegmentSize> 17308 </textSegmentSize>
       <dataSegmentSize> 488 </dataSegmentSize>
       <arguments> -m-v-analize-azconnect-azpost </arguments>
       <numArguments> 1-4 </numArguments>
       <size> 95188 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=nxt>
       <md5> 024c02fda0b3ef72975fe7f4d600ea75 </md5>
       <textSegmentSize> 5917 </textSegmentSize>
       <dataSegmentSize> 1104 </dataSegmentSize>
       <arguments> -t-s </arguments>
       <numArguments> 2-4 </numArguments>
       <size> 19458 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=pam>
       <md5> f12ce5a9fc714c6e8c2454cb1a274c37 </md5>
       <textSegmentSize> 4703 </textSegmentSize>
       <dataSegmentSize> 884 </dataSegmentSize>
       <arguments> -h-p </arguments>
       <numArguments> 1-2 </numArguments>
       <size> 17081 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=phorum>
       <md5> 6c1c7939bd91e17b8534abccc4f2c9d1 </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 0 </numArguments>
       <size> 2907 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=picobof>
       <md5> 3df483d4809936e885e7fd9dc6a54166 </md5>
       <textSegmentSize> 5058 </textSegmentSize>
       <dataSegmentSize> 308 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2-3 </numArguments>
       <size> 16449 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=priv8>
       <md5> 7196f50ff2b0c806cf549add4c7278db </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> -h-p-t-o </arguments>
       <numArguments> 4 </numArguments>
       <size> 3717 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=proftpd>
       <md5> e4a71ee5629a744abafc1b7e101b90bd </md5>
       <textSegmentSize> 11436 </textSegmentSize>
       <dataSegmentSize> 884 </dataSegmentSize>
       <arguments> -d-u-p-l-h-o-t </arguments>
       <numArguments> 7 </numArguments>
       <size> 25837 </size>
</ProcessSignature>
```

```
<ProcessSignature rdf:ID=qex>
       <md5> 4ff02f4c30707d56e88799bbe44e9ec6 </md5>
       <textSegmentSize> 5631 </textSegmentSize>
       <dataSegmentSize> 400 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 3 </numArguments>
       <size> 17496 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=realown>
       <md5> 338fe8fa41a7cd1fbbb5cdf72ee66f1d </md5>
       <textSegmentSize> 3185 </textSegmentSize>
       <dataSegmentSize> 6800 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 21791 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=remo>
       <md5> 3e505342cdf4892297e405c61cb574f5 </md5>
       <textSegmentSize> 4116 </textSegmentSize>
       <dataSegmentSize> 2248 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 17741 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=rootdown>
       <md5> 97ace86f8075e71c2e306e393012020f </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> -h-c-i-p-r-v </arguments>
       <numArguments> 2-6 </numArguments>
       <size> 12416 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=rpcexec>
       <md5> 1ae30ab01b2cd2ba7b7eb3e0cf37adf3 </md5>
       <textSegmentSize> 4032 </textSegmentSize>
       <dataSegmentSize> 2397 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 17744 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=smurf4>
       <md5> 5e62a17140d5198587044c90c79d5b88 </md5>
       <textSegmentSize> 4346 </textSegmentSize>
       <dataSegmentSize> 432 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 5 </numArguments>
       <size> 16356 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=spxeneo>
       <md5> 9752ef6c1df794be31bc985d5faca28c </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 1399 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=ss>
       <md5> 1034ee8bece356c5370d39673a3fef1a </md5>
       <textSegmentSize> 0 </textSegmentSize>
       <dataSegmentSize> 0 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 1 </numArguments>
       <size> 17215 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=sslbomb>
       <md5> cc7bdc36853ee5fae400cde398a22a8c </md5>
       <textSegmentSize> 4142 </textSegmentSize>
       <dataSegmentSize> 1720 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2-3 </numArguments>
       <size> 4076 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=symanhoddos>
       <md5> 7703f81e3af39355c943f5917a9f6b58 </md5>
       <textSegmentSize> 4288 </textSegmentSize>
       <dataSegmentSize> 324 </dataSegmentSize>
       <arguments> -fi-tp-ti-n </arguments>
       <numArguments> 2-4 </numArguments>
       <size> 15852 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=telexp>
       <md5> 64c9619b45ef671a765ac9da6bc373d3 </md5>
       <textSegmentSize> 3573 </textSegmentSize>
       <dataSegmentSize> 956 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 3 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=tlsdos>
       <md5> 79c150d47e2bd0485552f710c588d899 </md5>
       <textSegmentSize> 4110 </textSegmentSize>
       <dataSegmentSize> 1720 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 2 </numArguments>
       <size> 3 </size>
</ProcessSignature>
```


Sophos Ex. 1007
Page 187 of 216

```
<ProcessSignature rdf:ID=vwexploit>
       <md5> fc0e8ed87f104ae8c93d7a886d752d70 </md5>
       <textSegmentSize> 3514 </textSegmentSize>
       <dataSegmentSize> 2516 </dataSegmentSize>
       <arguments> null </arguments>
       <numArguments> 4 </numArguments>
       <size> 17686 </size>
</ProcessSignature>

<ProcessSignature rdf:ID=yapping>
       <md5> cd9cb37c44a832d0661ec61dcdedab23 </md5>
       <textSegmentSize> 2496 </textSegmentSize>
       <dataSegmentSize> 312 </dataSegmentSize>
       <arguments> -h-s-c </arguments>
       <numArguments> 3 </numArguments>
       <size> 14137 </size>
</ProcessSignature>
```

Appendix F: Network signatures

The table expanding the following pages corresponds to the network signatures of the programs surveyed in this project.

Sophos Ex. 1007
Page 189 of 216

```
<TrafficSignature rdf:ID=apache-nose>
    <packetCounter> 3 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 0 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 30 </deviation>
    <flags> "0,8,1,5,4,5,4,0,0,0,0,0,0,0,0,0" </flags>
    <lengths> "0,0,142,298,0,0,0,0,0,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,1132,2924,0,0,0,0,0,0,0,0,0,0,0" </hashes>
    <execVariant> 1a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=apache-nose>
    <packetCounter> 8 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 23 </destinationPort1>
    <destinationPort2> 9191 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 30 </deviation>
    <flags> "0,8,1,1,1,1,8,1,0,0,0,0,0,0,0,0" </flags>
    <lengths> "0,0,0,0,1032,0,0,0,0,0,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,0,0,1950,0,0,0,0,0,0,0,0,0,0,0" </hashes>
    <execVariant> 1b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=apache-scalp>
    <packetCounter> 16 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 0 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,1,1,1,1,1,1,1,1,1,1,1,1,1,1" </flags>
    <lengths> "0,0,0,0,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448" </lengths>
    <hashes> "0,0,0,0,2138,1950,1950,1950,1950,2013,1264,1950,1950,1950,1950,1950" </hashes>
    <execVariant> 2a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=apache-scalp>
    <packetCounter> 9 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 22 </destinationPort1>
    <destinationPort2> 0 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,1,1,1,1,1,1,1,0,0,0,0,0,0,0" </flags>
    <lengths> "0,0,0,1448,1448,1448,0,1448,1448,0,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,0,2138,1950,1950,0,1950,1950,0,0,0,0,0,0,0" </hashes>
    <execVariant> 2b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=apache-scalp>
    <packetCounter> 16 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 0 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,1,5,1,1,1,1,1,1,1,1,1,1,1,1" </flags>
    <lengths> "0,0,0,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448,1448" </lengths>
    <hashes> "0,0,0,2143,1950,1950,1950,1166,1950,1950,1950,1950,1950,1950,1950,1950" </hashes>
    <execVariant> 2c </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=atphttp>
    <packetCounter> 10 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 65535 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,1,1,7,8,0,0,1,1,0,0,0,0,0,0" </flags>
    <lengths> "0,0,0,820,0,0,0,0,96,1232,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,0,1928,0,0,0,0,1562,1674,0,0,0,0,0,0" </hashes>
    <execVariant> 3a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=bazooka>
    <packetCounter> 5 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 0 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,8,8,8,0,0,0,0,0,0,0,0,0,0,0" </flags>
    <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
    <execVariant> 4a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=bazooka>
    <packetCounter> 8 </packetCounter>
    <fragmentation> 0 </fragmentation>
    <sourceAddress> -8205 </sourceAddress>
    <sourcePort> 0 </sourcePort>
    <destinationAddress> -5110 </destinationAddress>
    <destinationPort1> 80 </destinationPort1>
    <destinationPort2> 65535 </destinationPort2>
    <windowSize> 0 </windowSize>
    <deviation> 0 </deviation>
    <flags> "0,8,1,1,7,8,0,0,0,0,0,0,0,0,0,0" </flags>
    <lengths> "0,0,0,820,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
    <hashes> "0,0,0,1928,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
    <execVariant> 4b </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=exceed>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 22 </destinationPort1>
       <destinationPort2> 110 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,1,0,8,1,1,1,1,1,1" </flags>
       <lengths> "0,0,0,1448,1448,89,0,0,0,0,0,0,6,100,182,0" </lengths>
       <hashes> "0,0,0,2576,1950,2312,0,0,0,0,0,0,349,1512,977,977" </hashes>
       <execVariant> 5a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=exceed>
       <packetCounter> 8 </packetCounter>
       <fragmentation> 1 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,7,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,1448,1448,89,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,2576,1950,2312,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 5b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=numby>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 110 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,7,8,1,1,1,1,1,1,1" </flags>
       <lengths> "0,0,0,266,0,0,190,0,0,0,0,6,100,182,0,33" </lengths>
       <hashes> "0,0,0,2838,0,0,1334,0,0,0,0,349,1306,1483,0,0" </hashes>
       <execVariant> 6a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=numby>
       <packetCounter> 5 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 22 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,4,1,1,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,266,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,2838,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 6b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=pam>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 6023 </destinationPort1>
       <destinationPort2> 110 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,8,8,8,8,8,8,1,1,1,1,1,1,1,1" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,6,100,182,0,33,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,349,1223,1617,0,1689,1689" </hashes>
       <execVariant> 7a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=pam>
       <packetCounter> 7 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 6022 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,8,8,8,8,8,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 7b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=realown>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,8,1,1,1,1,8,1,1,1,1" </flags>
       <lengths> "0,0,0,34,0,0,0,0,39,0,0,0,0,48,0,0" </lengths>
       <hashes> "0,0,0,1759,0,0,0,0,1868,0,0,0,0,1814,0,0" </hashes>
       <execVariant> 8a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=realown>
       <packetCounter> 5 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 19 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,8,8,8,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 8b </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=qex>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,27,0,538,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,286,0,839,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 9a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=qex>
        <packetCounter> 14 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 110 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,1,1,1,1,1,1,0,0" </flags>
        <lengths> "0,0,0,1448,1448,1448,1448,664,0,0,27,0,23,0,0,0" </lengths>
        <hashes> "0,0,0,2432,2484,2484,2484,2651,0,0,1720,0,1105,0,0,0" </hashes>
        <execVariant> 9b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=telexp>
        <packetCounter> 9 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 30 </deviation>
        <flags> "0,8,1,1,1,1,1,1,1,1,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1448,1448,0,1448,1448,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2432,2484,0,2484,2484,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 10a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=telexp>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 110 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 30 </deviation>
        <flags> "0,8,1,1,1,1,7,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,7,7,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,458,451,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 10b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=vwexploit>
        <packetCounter> 5 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -2322 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,4,4,8,0,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 11a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=305monit>
        <packetCounter> 7 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 23 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1314,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,3256,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 12a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=305monit>
        <packetCounter> 7 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1414,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,3256,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 12b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=akill2>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,7,0,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,462,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,1771,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 13a </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=anubisexp>
       <packetCounter> 12 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 23 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 30 </deviation>
       <flags> "0,8,5,1,5,1,5,1,1,1,1,0,0,0,0,0" </flags>
       <lengths> "0,0,0,27,538,3,3,5,0,0,163,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,386,23,23,32,83,0,0,213,0,0,0,0,0" </hashes>
       <execVariant> 14a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=arksink>
       <packetCounter> 11 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 110 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,7,0,0,8,1,7,1,0,0,0,0,0" </flags>
       <lengths> "0,0,0,18,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,939,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 15a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=bysin>
       <packetCounter> 6 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -9720 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,7,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,33,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,2104,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 16a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=cge>
       <packetCounter> 2 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 2812 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 17a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=cobain>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,7,8,1,1,1,1,1,7,8" </flags>
       <lengths> "0,0,0,90,47,0,1448,133,0,0,90,0,47,1448,133,0" </lengths>
       <hashes> "0,0,0,2363,2343,0,2344,2823,0,0,2363,0,2343,2344,2823,2823" </hashes>
       <execVariant> 18a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=delefate>
       <packetCounter> 6 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -2897 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 22 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,1448,490,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,2434,2655,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 19a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=emule>
       <packetCounter> 13 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,2,2,2,2,2,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,23,234,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,134,2234,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 20a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=esignal>
       <packetCounter> 11 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 143 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,8,1,1,1,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,50,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,2239,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 21a </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=ethboom>
        <packetCounter> 9 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,7,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1448,1448,1448,381,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2773,3360,3360,3360,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 22a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=extremail2>
        <packetCounter> 6 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,8,8,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,558,550,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 23a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=ezploit>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 8080 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,2,7,7,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 24a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=fping>
        <packetCounter> 5 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 6004 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,0,0,1,0,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 25a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=freezer>
        <packetCounter> 7 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 1 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1308,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2728,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 26a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=fspscan>
        <packetCounter> 10 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,1448,1448,1186,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,2748,3330,3330,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 27a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=fspscan>
        <packetCounter> 9 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,7,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1448,1448,1186,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2748,3330,3330,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 27b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=gtkftpd>
        <packetCounter> 14 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 21 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,8,8,8,1,1,1,7,8,1,1,1,7" </flags>
        <lengths> "0,0,0,32,0,0,0,0,32,0,0,0,0,32,0,0" </lengths>
        <hashes> "0,0,0,210,0,0,0,0,210,0,0,0,0,213,0,0" </hashes>
        <execVariant> 28a </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=iawebmail>
       <packetCounter> 2 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -092 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 445 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 29a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=ircd>
       <packetCounter> 16 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 21 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,0,9,9,9,1,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,255,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,2523,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 30a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=isspam>
       <packetCounter> 3 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 515 </destinationPort1>
       <destinationPort2> 7788 </destinationPort2>
       <windowSize> 1 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,8,0,1,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 31a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=lsasrv>
       <packetCounter> 11 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,1,1,1,1,0,0,0,0,0" </flags>
       <lengths> "0,0,0,27,538,3,3,5,0,0,163,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,286,839,6,5,349,0,0,1713,0,0,0,0,0" </hashes>
       <execVariant> 32a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=namedsploit>
       <packetCounter> 14 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,7,8,1,1,1,1,1,7,8" </flags>
       <lengths> "0,0,0,90,47,0,1448,133,0,0,90,1448,180,0,0,0" </lengths>
       <hashes> "0,0,0,2363,2343,0,2344,2823,0,0,2363,2343,2910,0,0,0" </hashes>
       <execVariant> 33a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=niprint>
       <packetCounter> 2 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 53 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,7,0,0,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 34a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=nptd>
       <packetCounter> 10 </packetCounter>
       <fragmentation> 0 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 22 </destinationPort1>
       <destinationPort2> 65535 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,1,1,1,1,1,1,8,1,0,0,0,0" </flags>
       <lengths> "0,0,0,5,0,1448,1448,1448,252,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,532,0,3188,3360,3360,1520,0,0,0,0,0,0,0" </hashes>
       <execVariant> 35a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=nxt>
       <packetCounter> 4 </packetCounter>
       <fragmentation> 1 </fragmentation>
       <sourceAddress> -8205 </sourceAddress>
       <sourcePort> 0 </sourcePort>
       <destinationAddress> -5110 </destinationAddress>
       <destinationPort1> 80 </destinationPort1>
       <destinationPort2> 0 </destinationPort2>
       <windowSize> 0 </windowSize>
       <deviation> 0 </deviation>
       <flags> "0,8,1,1,3,0,0,0,0,0,0,0,0,0,0,0" </flags>
       <lengths> "0,0,0,5,0,0,0,0,0,0,0,0,0,0,0" </lengths>
       <hashes> "0,0,0,532,0,0,0,0,0,0,0,0,0,0,0" </hashes>
       <execVariant> 36a </execVariant>
</TrafficSignature>
```


Sophos Ex. 1007
Page 195 of 216

```
<TrafficSignature rdf:ID=nxt>
        <packetCounter> 11 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 65535 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,1,8,1,0,0,0,0,0" </flags>
        <lengths> "0,0,0,5,0,1448,1448,1200,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,532,0,3188,3360,3360,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 36b </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=phorum>
        <packetCounter> 2 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 21 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,8,0,0,0,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 37a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=picobof>
        <packetCounter> 7 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 65535 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,7,7,0,7,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,234,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,4023,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 38a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=priv8>
        <packetCounter> 2 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 135 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 39a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=rootdown>
        <packetCounter> 12 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 23 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 1 </windowSize>
        <deviation> 30 </deviation>
        <flags> "0,8,1,1,1,1,1,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,201,0,802,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2352,0,234,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 40a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=rpcexec>
        <packetCounter> 16 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 1 </destinationPort1>
        <destinationPort2> 2 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 41a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=smurf4>
        <packetCounter> 7 </packetCounter>
        <fragmentation> 1 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,3,1,1,1,6,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,132,0,1308,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,2145,0,913,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 42a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=spxeneo>
        <packetCounter> 9 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 22 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 1 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,0,1,0,1,1,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,308,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,925,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 43a </execVariant>
</TrafficSignature>
```

```
<TrafficSignature rdf:ID=ss>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 6080 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,8,8,8,8,8,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 44a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=sslbomb>
        <packetCounter> 15 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 1 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,7,8,1,1,1,7,8,1,1,1,7" </flags>
        <lengths> "0,0,0,31,0,0,0,0,31,0,0,0,0,31,0,0" </lengths>
        <hashes> "0,0,0,2101,0,0,0,0,2102,0,0,0,0,2103,0,0" </hashes>
        <execVariant> 45a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=symanhoddos>
        <packetCounter> 16 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 617 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,8,1,1,1,1,1,1,1,1,7,1,0,0" </flags>
        <lengths> "0,0,0,0,0,1448,1448,1448,1448,1448,1448,1441,0,8,0,0" </lengths>
        <hashes> "0,0,0,0,0,3360,3360,3360,3360,3360,3360,3360,0,103,0,0" </hashes>
        <execVariant> 46a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=tlsdos>
        <packetCounter> 8 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 23 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,8,8,1,1,1,1,1,1,1,1,1,1,1,1" </flags>
        <lengths> "0,0,0,203,203,203,203,203,203,203,203,203,203,203,203" </lengths>
        <hashes> "0,0,0,3138,3950,3950,5950,5950,3950,1012,1265,1221,1221,1221,1221,1221" </hashes>
        <execVariant> 47a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=yapping>
        <packetCounter> 16 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 22 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,8,1,1,1,1,1,1,1,1,1,1,1,1" </flags>
        <lengths> "0,0,0,0,0,0,22,544,24,0,416,0,16,48,0,64" </lengths>
        <hashes> "0,0,0,0,0,0,1528,1728,104,0,1323,0,43,1774,0,0" </hashes>
        <execVariant> 48a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=proftpd>
        <packetCounter> 9 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 25 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,1,1,7,1,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,0,16,26,1309,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,0,1260,1936,1963,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 49a </execVariant>
</TrafficSignature>

<TrafficSignature rdf:ID=remo>
        <packetCounter> 10 </packetCounter>
        <fragmentation> 0 </fragmentation>
        <sourceAddress> -8205 </sourceAddress>
        <sourcePort> 0 </sourcePort>
        <destinationAddress> -5110 </destinationAddress>
        <destinationPort1> 80 </destinationPort1>
        <destinationPort2> 0 </destinationPort2>
        <windowSize> 0 </windowSize>
        <deviation> 0 </deviation>
        <flags> "0,8,1,1,1,1,7,0,0,0,0,0,0,0,0,0" </flags>
        <lengths> "0,0,0,1448,1448,1218,0,0,0,0,0,0,0,0,0,0" </lengths>
        <hashes> "0,0,0,1878,1890,1890,0,0,0,0,0,0,0,0,0,0" </hashes>
        <execVariant> 50a </execVariant>
</TrafficSignature>
```


Sophos Ex. 1007
Page 197 of 216

Bibliography

[1] Abad, C., Taylor, J., Sengul, C., and Yurcik, W. *Log Correlation for Intrusion Detection: A Proof of Concept.* 19th Annual Computer Security Applications Conference, vol. 1, ACSAC, pp. 255–265, Las Vegas, NV, December 2003.

[2] Allen, J., Christie, A., Fithen, W., McHugh, J., Pickel, J., and Stoner, E. *State of the Practice of Intrusion Detection Technologies.* Tech. Rep. 99-TR-028, ESC-99-028, Carnegie Mellon University, Software Engineering Institute, CMU/SEI, Pittsburgh, PA, 2000.

[3] Alonso, E. *AI and Agents: State of the Art.* The AI Magazine, pp. 25–29, vol. 23, num. 3, 2000.

[4] Amoroso, E. G. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, and Reponse.* Intrusion.net Books. February 1999.

[5] Anderson, D., Frivold, T., Tamaru, A., and Valdes, A. *Next-generation Intrusion Detection Expert System, NIDES.* Tech. Rep. SRI–CSL–95–07, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1994.

[6] Anderson, J. P. *Computer Security threat Monitoring and Surveillance.* Tech. rep., James P. Anderson Company, Fort Washington, PA, Abril 1980.

[7] Antoniou, G., and van Harmelen, F. *Web Ontology Language: OWL.* Tech. rep., Department of Computer Science, University of Crete, and Department of Artificial Intelligence, Vrije Universitiet, 2003.

[8] Anzen. *AFJ Intrusion Detector*, Anzen Corp., June 2001. URL: http://www.anzen.comp/products/afj (on-line resource).

[9] Apap, F., Honig, A., Hershkop, S., Eskin, E., and Stolfo, S., Eds. *Detecting Maliciuos Software by Monitoring Anomalous Windows Registry Accesses* (October 2002), LNCS 2516, Recent Advances in Intrusion Detection, Springer.

[10] Arbaugh, W. A., Fithen, W. L., and McHugh, J. *Windows of Vulnerability: A Case Study Analysis.* IEEE Computer, pp. 52–59, num. 1, 2000.

[11] Asaka, M., Tsuchiya, M., Onabuta, T., Okasawa, S., and Goto, S. *Local Attack Detection and Intrusion Route Tracing.* IEICE Trans. Commun. New Paradigms in Network Management, pp. 1826–1833, vol. E82-B, num. 11, 1999.

[12] Avizienis, A., Laprie, J., and Randell, B. *Fundamental Concepts of Dependability.* Tech. Rep. N01145, LAAS-CNRS, April 2001.

[13] Axelsson, S. *Research in Intrusion-Detection Systems: A Survey.* Tech. Rep. TR: 98-17, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweeden, 1998.

[14] Ayala, G. *Intelligent Agents for a Lifelong Learning Environment in Information Technologies.* International Journal of Continuing Engineering Education and Life-Long Learning, pp. 189–200, vol. 12, num. 1-4, 2000.

[15] Balasubramaniyan, J. S., Garcia-Fernandez, J. O., Isacoff, D., Spafford, E. H., and Zamboni, D. *An Architecture for Intrusion Detection using Autonomous Agents.* 14th Anual Computer Security Applications Conference, pp. 13–24, 1998.

[16] Bass, T. *Intrusion Detection Systems and Multisensor Data Fusion.* Communications of the ACM, pp. 99–105, vol. 1, num. 1, 1999.

[17] Bellifemine, F., Poggi, A., and Rimassa, G. *JADE - A FIPA-compliant Agent Framework.* PAAM '99, pp. 97–108, London, UK, 1999.

[18] Bishop, M. *Computer Security: Art and Science*, 1st ed. Addison-Wesley Pub. Co. December 2002.

[19] Borselius, N. *Mobile Agent Security.* Electronics & Communication Engineering Journal, pp. 211–218, vol. 14, num. 5, 2002.

[20] Breiling, S., Plato, A., and Shulak, B. *BlackICE Sentry Signature Reference Guide.* Internet Security Systems, ISS, June 2001.

[21] Brena, R., Aguirre, J. L., and Trevino, A. C. *Just-in-time Information and Knowledge: Agent Technology for KM Bussiness Processes.* 2001 IEEE Systems, Man, and Cybernetics Conference, September 2001.

[22] Bridges, S. M., and Vaughn, R. B. *Intrusion Detection via Fuzzy Data Mining.* 12th Annual Canadian Information Technology Security Symposium, pp. 109–122, Ottawa, Canada, June 2000.

[23] Brugali, D., and Sycara, K. *Towards Agent Oriented Application Frameworks.* ACM Computing Surveys, pp. 21–26, vol. 32, num. 1, 2000.

[24] Bugtraq. *Bugtraq Vulnerability Repository*, Securityfocus.com, 2003. URL: http://www.securityfocus.com (on-line resource).

[25] Bussmann, S., and Müller, J. *A Communication Architecture for Cooperating Agents.* Computers and Artificial Intelligence, pp. 37–53, vol. 12, num. 1, 1993.

[26] CA. *E-Trust Intrusion Detection*, Computer Associates, Inc., December 2003. URL: http://www3.ca.com/ (on-line resource).

[27] Cannady, J. *Artificial Neural Networks for Misuse Detection.* National Information Systems Security Conference, NISSC, no. 1, pp. 443–456, Arlington, VA, October 1998.

[28] Cansian, A. M., de Souza, A. Z. T., Filho, S. L., and Moreira, E. S. *A Password-capture System for the Security of Computer Networks.* Department of Computer Science and Statistics, UNESP, Brazil, 1999.

[29] Captus Networks. *CaptIO and CaptIO-G*, Captus Networks, Inc., May 2004. URL: http://www.captusnetworks.com/ (on-line resource).

[30] Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., and Wilkinson, K. *Jena: Implementing the Semantic Web Recommendations.* Tech. Rep. HPL-2003-146, HP Laboratories Bristol, Bristol, UK, December 2003.

[31] Ceballos-Cancino, H. G. *Ontology Administration in Multiagent Environments using an Ontology Agent applied to JITIK (in Spanish).* Master's thesis, ITESM–Monterrey. Monterrey, Mexico, 2003.

[32] CERT. *Statistics of the Computer Emergency Response Team Coordination Center, CERT–CC*, Carnegie Mellon University, 2004. URL: http://www.cert.org/stats/ (on-line resource).

[33] Christey, S. M., and Martin, R. A. *A Progress Report on the CVE Initiative.* 14th Annual Computer Security Incident Handling Conference, Kona, HA, June 2002.

[34] Christoph, G. G., Jackson, K. A., Neuman, M. C., Siciliano, C. L., Simmonds, D. D., Stallings, C. A., and Thopson, J. L. *UNICORN: Misuse Detection for UNICOS.* ACM/IEEE Supercomputing Conference, no. 56, ACM Press and IEEE Computer Society Press, December 1995.

[35] CISCO. *The Abacus Project*, CISCO Systems, Inc., June 2001. URL: http://www.digital.com/info/security/id.htm (on-line resource).

[36] CISCO. *Net Ranger*, CISCO Systems, Inc., January 2004. URL: http://www.cisco.com/univercd/cc/td/doc/product/iaabu/netrangr/ (on-line resource).

[37] Coit, C. J., Staniford, S., and McAlerney, J. *Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort.* DARPA Information Survivability Conference and Exposition (DISCEX II'01), vol. 1, Anaheim, CA, June 2001.

[38] Comer, D. E., and Stevens, D. L. *Internetworking with TCP/IP series*, 1st ed. Prentice Hall. Upper Saddle River, NJ, 1997.

[39] Coull, S., Branch, J., Szymanski, B., and Breimer, E. *Intrusion Detect: A Bioinformatics Approach.* 19th Annual Computer Security Applications Conference, pp. 24–33, Las Vegas, NV, December 2003.

[40] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. *Formatguard: Automatic Protection from Printf Format String Vulnerabilities*. USENIX Security Symposium, pp. 191–199, Washington, DC, August 2001.

[41] Crosbie, M., and Spafford, E. H. *Defending a Computer System using Autonomous Agents*. 18th National Information Systems Security Conference, pp. 549–558, October 1995.

[42] Curry, D., Klaus, C., Ranum, M., Spafford, E., and Sutterfieldi, L. *CSI Roundtable: Experts Discuss Present and Future of Intrusion Detection Systems*. Computer Security Journal, vol. 14, num. 1, 1998.

[43] Curry, D. A., Debar, H., and Feinstein, B. S. *Intrusion Detection Message Exchange Format*. Intrusion Detection Working Group – Internet Engineering Task Force, Internet Draft, January 2004.

[44] CyberTrace. *CyberTrace Network Security Management*, CyberTrace, Inc., February 2004. URL: http://www.cybertrace.com/ctids.html (on-line resource).

[45] Dasgupta, P., Moser, L. E., and Melliar-Smith, P. M. *The Security Architecture for MAgNET: A Mobile Agent E-commerce System*. 3rd International Conference on Telecommunications and E-commerce, pp. 289–298, Dallas, TX, November 2000.

[46] Debar, H., Becker, M., and Siboni, D. *A Neural Network Component for an Intrusion Detection System*. IEEE Computer Society Symposium on Research in Security and Privacy, Los Alamitos, CA, pp. 240–250, Oakland, CA, May 1992.

[47] DEC. *POLICENTER Security Compliance Manager*, Digital Equipment Corporation, Inc., August 1994. URL: http://www.digital.com/info/security/id.htm (on-line resource).

[48] Deloitte and Touche. *2003 Global Security Survey*, Deloitte and Touche, Inc., May 2003. URL: http://www.deloitte.com/dtt/cda/doc/content (on-line resource).

[49] Denning, D. E. *An Intrusion Detection Model*. IEEE Transactions on Software Engineering, no. 2 in 13, pp. 222–232, February 1987.

[50] Denning, D. E. *A Prototype IDES: A Real-time Intrusion Detection Expert System*. Tech. rep., Computer Science Laboratory, SRI International, Menlo Park, CA, August 1987.

[51] Deswarte, Y., Blain, L., and Fabre, J.-C. *Intrusion Tolerance in Distributed Computing Systems*. IEEE Symposium on Security and Privacy, pp. 110–121, 1991.

[52] DOE-CIAC. *DOE-CIAC's Network Intrusion Detector (NID)*, National Nuclear Security Administration, April 2004. URL: http://ciac.llnl.gov/cstc/nid/nid.html (on-line resource).

[53] Dowel C, R. *The Computer Watch Data Reduction Tool*. 13th National Computer Security Conference, pp. 99–108, Washington, DC, October 1990.

[54] Durfee, E. H., and Rosenschein, J. *Distributed Problem Solving and Multiagent Systems: Comparisons and Examples.* 13th International Workshop on DAI, M. Klein, Ed., pp. 94–104, Lake Quinalt, WA, 1994.

[55] Edwards, M. J. *Kane Security Monitor 3.03.* Windows Network and .NET Magazine, num. 1, 1998 (on-line resource.

[56] Endler, D. *Intrusion Detection Applying Machine Learning to Solaris Audit Data.* Annual Computer Security Applications Conference, ACSAC, IEEE Computer Society Press, pp. 268–279, Scottsdale, AZ, 1998.

[57] EnGarde. *T-sight: On Target Security*, ENGARDE Systems, April 2004. URL: http://www.engarde.com/software/t-sight/ (on-line resource).

[58] Enterasys. *Dragon Intrusion Detection Defense System*, Enterasys Intrusion Defense, Inc., April 2004. URL: http://www.enterasys.com/products/ids/ (on-line resource).

[59] Erlingsson, and Schneider. *SASI Enforcement of Security Policies: A Retrospective.* WNSP: New Security Paradigms Workshop, ACM Press, pp. 87–95, 2000.

[60] Escamilla, T. *Intrusion Detection: Network Security Beyond the Firewall.* John Wiley & Sons, Inc. New York, NY, 1998.

[61] Exploit Database. *Full-disclosure Exploit Database*, Black Sheep Networks, Inc., 2003. URL: http://www.exploitdatabase.com (on-line resource).

[62] Farmer, W. M., Guttman, J. D., and Swarup, V. *Security for Mobile Agents: Authentication and State Appraisal.* 4th European Symposium on Research in Computer Security, pp. 118–130, Rome, Italy, 1996.

[63] Fedoruk, A., and Deters, R. *Improving Fault-Tolerance by Replicating Agents.* 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, pp. 737–744, Bologna, Italy, July 15–19 2002.

[64] Forrest, S., Hofmer, S. A., Somayaji, A., and Longstaff, T. A. *A Sense of Self for UNIX Processes.* IEEE Symposium on Security and Privacy, IEEE Press, Oakland, CA, May 1996.

[65] Frank, J. *Artificial Intelligence and Intrusion Detection: Current and Future Directions.* 17th National Computer Security Conference, pp. 22–33, October 1994.

[66] Frincke, D., and Ho, Y. *Planning, Petri Nets, and Intrusion Detection.* 21st National Information Systems Security Conference, pp. 346–361, October 1998.

[67] Garfinkel, S., and Spafford, E. H. *Practical UNIX Security*, 2nd ed. O'Reilly. April 1996.

[68] Garvey, T. D., and Lunt, T. F. *Model-based Intrusion Detection.* 14th National Computer Security Conference, NIST, National Institute of Standards and Technology/National Computer Security Center, pp. 372–385, Baltimore, MD, October 1998.

[69] GDataMine. *Gnome Data Mining Tool*, GNOME, August 2003. URL: http://www.togaware.com/datamining/gdatamine (on-line resource).

[70] GFI. *LANguard*, GFI Security, March 2004. URL: http://www.gfi.com/languard/ (on-line resource).

[71] Giffin, J. T., Jha, S., and Miller, B. P. *Efficient Context-Sensitive Intrusion Detection*. 11th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2004.

[72] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. *A Secure Environment for Untrusted Helper Applications*. 6th USENIX Security Symposium, pp. 1–13, San Jose, CA, July 1996.

[73] Goldschlag, D., Reed, M., and Syverson, P. *Onion Routing for Anonymous and Private Internet Connections*. Communications of the ACM, pp. 39–41, vol. 42, num. 2, 1999.

[74] Guan, X., Yang, Y., and You, J. *POM – A Mobile Agent Security Model against Malicious Hosts*. 4th International Conference on High-Performance Computing in the Asia-Pacific Region, vol. 2, pp. 1165–1166, May 2000, Beijing, China.

[75] Habra, N., Charlier, B. L., Mounji, A., and Mathieu, I. *ASAX: Software Arquitecture and Rule-based Language for Universal Audit Trail Analysis*. European Symposium on Research in Computer Security, ESORICS, pp. 435–450, num. 1, 1992.

[76] Halme, L. R., and Bauer, K. R. *A Taxonomy of Anti-intrusion Techniques*. 18th National Information Systems Security Conference, pp. 163–172, Baltimore, MD, October 1995.

[77] Heberlein, T., Dias, G., Levitt, K., Wood, B. M. J., and Wolber, D. *A Network Security Monitor*. IEEE Symposium on Research in Security and Privacy, IEEE Press, pp. 296–304, Los Alamitos, CA, May 1990.

[78] Helmer, G. G., Wong, J., Honovar, V., and Miller, L. *An Architecture for Intrusion Detection using Autonomous Agents*. Tech. rep., Department of Computer Science, Iowa State University, Des Moines, IA, 1999.

[79] Hewlett Packard. *HP IDS 9000*, Hewlett Packard, Inc., June 2001. URL: http://www.hp.com/security/products/ids (on-line resource).

[80] Hirata, K. *A Classification of Abduction: Abduction for Logic Programming*. Machine Intelligence, Oxford University Press, pp. 397–424, vol. 1, num. 14, 1995.

[81] Hochberg, J., Jackson, K., Stallings, C., McClary, J. F., DuBois, D., and Ford, J. *NADIR: An Automated System for Detecting Network Intrusion and Misuse*. Elsevier Advanced Technology Publications, pp. 235–248, vol. 12, num. 3, 1993.

[82] Hofmeyr, S. A., and Forrest, S. *Architecture for an Artificial Immune System.* Evolutionary Computation, pp. 443–473, vol. 8, num. 4, 2000.

[83] Hofmeyr, S. A., Forrest, S., and Somayaji, A. *Intrusion Detection using Sequences of System Calls.* Journal of Computer Security, pp. 151–180, vol. 6, num. 1, 1998.

[84] Houle, K. H., and Weaver, G. M. *Trends in Denial of Service Attack Technology.* Tech. rep., CERT Coordination Ceter, Pittsburgh, PA, October 2001.

[85] Huagang, X. *Build a Secure System with LIDS.* LinuxSecurity.com, num. 1, 2000 (on-line resource).

[86] Huhns, M. N., and Singh, M. P. *Agents and Multiagent systems: Themes, Approaches, and Challenges.* Readings in Agents, pp. 1–23, num. 1, 1998.

[87] Ilgun, K. *USTAT: A Real-Time Intrusion Detection System for UNIX.* IEEE Symposium on Security and Privacy, pp. 16–28, Oakland, CA, May 1993.

[88] Intouch INSA. *Network Security Agent*, Intouch INSA, February 2004. URL: http://www.ttinet.com (on-line resource).

[89] ISS. Internet Security Systems, Inc., July 2003. URL: http://www.iss.net (on-line resource).

[90] Jackson, K. A. *Intrusion Detection System (IDS) Product Survey.* Tech. Rep. LA-UR-99-3883, Los Alamos National Laboratory, Los Alamos, NM, June 1999.

[91] Jackson, K. A., DuBois, D. H., and Stallings, C. A. *An Expert System Application for Network Intrusion Detection.* 14th National Computer Security Conference, National Institute of Standards and Technology/National Computer Security Center, pp. 215–225, Washington, DC, October 1991.

[92] Janakiraman, R., Waldvogel, M., and Zhang, Q. *INDRA: A Peer-to-Peer Approach to Network Intrusion Detection and Prevention.* IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE, June 2003.

[93] Jansen, W., Mell, P., Karygiannis, T., and Marks, D. *Mobile Agents in Intrusion Detection and Response.* 12th Annual Canadian Information Technology Security Symposium, Ottawa, Canada, June 2000.

[94] Jensen, F. V. *Bayesian Networks and Decision Graphs.* Springer Verlag Ed. Berlin, Germany, 2001.

[95] Jou, Y. F., Gong, F., Sargor, C., Wu, S. F., and Cleaveland, W. R. *Architecture Design of a Scalable Intrusion Detection System for the Emerging Network Infrastructure.* Tech. Rep. CDRLA-005, Department of Computer Science, North Carolina State University, Raleigh, NC, April 1997.

[96] JPCAP. *Java Packet Capture library*, Java Foundry, February 2003. URL: http://sourceforge.net/projects/jpcap/ (on-line resource).

[97] Kemmerer, R. A. *NSTAT: A Model-based Real-time Network Intrusion Detection System.* Tech. Rep. TR-CS-97-18, Reliable Software Group, Department of Computer Science, University of California-Santa Barbara, Santa Barbara, CA, November 1997.

[98] Kerschbaum, F., Spafford, E. H., and Zamboni, D. M. *Using Internal Sensors and Embedded Detectors for Intrusion Detection.* Journal of Computer Security, vol. 10, pp. 23–70, 2002.

[99] Kihlstrom, K. P., and Narasimhan, P. *The Starfish System: Providing Intrusion Detection and Intrusion Tolerance for Middleware Systems.* IEEE Workshop on Object-oriented Real-time Dependable Systems, Guadalajara, Mexico, January 2003.

[100] Kim, E. H., and Spafford, E. H. *The Design and Implementation of TRIPWIRE: A File System Integrity Checker.* 2nd ACM Conference on Computers and Communications Security, pp. 18–29, Fairfax, VA, November 1994.

[101] Klevinsky, T. J., Laliberte, S., and Gupta, A. *Hack I.T. – Security through Penetration Testing*, 1st ed. Addison-Wesley Longman. February 2002.

[102] Ko, C., Fink, G., and Levitt, K. *Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring.* 10th Annual Computer Security Applications Conference, IEEE Computer Society Press, pp. 134–144, Orlando, FL, 1994.

[103] Kumar, S. *Classification and Detection of Computer Intrusions.* PhD thesis, Department of Computer Sciences, Purdue University. West Lafayette, IN, 1995.

[104] Kumar, S., and Spafford, E. H. *An Application of Pattern Matching in Intrusion Detection.* Tech. Rep. CSD–TR–94–013, Department of Comptuter Sciences, Purdue University, West Lafayette, IN, June 1994.

[105] Kumar, S., and Spafford, E. H. *A Pattern Matching Model for Misuse Intrusion Detection.* 17th National Computer Security Conference, pp. 11–21, Baltimore, MD, October 1994.

[106] Kumar, S., and Spafford, E. H. *A Software Architecture to Support Misuse Intrusion Detection.* 18th National Information Security Conference, pp. 194–204, October 1995.

[107] Kuri, J., Navarro, G., M'e, L., and Heye, L. *A Pattern Matching Based Filter for Audit Reduction and Fast Detection of Potential Intrusions.* Recent Advances in Intrusion Detections conference (RAID '00), pp. 17–27, Berlin, Germany, 2000.

[108] Lane, T., and Brodley, C. E. *Temporal Sequence Learning for Anomaly Detection.* 5th ACM Conference on Computer and Communications Security, pp. 150–158, San Francisco, CA, 1998.

[109] Lane, T., and Brodley, C. E. *Temporal Sequence Learning and Data Reduction for Anomaly Detection.* ACM Transactions on Information Systems Security, pp. 295–331, vol. 2, num. 3, 1999.

[110] Lange, D. B., and Oshima, M. *Programming and Deploying Java Mobile Agents with Aglets.* Addison–Wesley Press. Menlo Park, CA, 1998.

[111] Lau, F., Rubin, S. H., Smith, M. H., and Trajovic, L. *Distributed Denial of Service Attacks.* IEEE International Conference on Systems, Man, and Cybernetics, pp. 2275–2280, Nashville, TN, October 2000.

[112] Lee, W., Nimbalkar, R. A., Yee, K. K., Patil, S. B., Desai, P. H., Tran, T. T., and Stolfo, S. J. *A Data Mining and CIDF Based Approach for Detecting Novel and Distributed Intrusions.* Lecture Notes in Computer Science, pp. 49–65, vol. 1907, num. 1, 2000.

[113] Lee, W., and Stolfo, S. *Data Mining Approaches for Intrusion Detection.* 7th USENIX Security Symposium, pp. 79–93, San Antonio, TX, 1998.

[114] Lee, W., Stolfo, S. J., and Mok, K. W. *A Data Mining Framework for Building Intrusion Detection Models.* IEEE Symposium on Security and Privacy, pp. 120–132, 1999.

[115] Lunt, T., Tamaru, A., Gilham, F., Jagannathan, R., Neumann, P., Javitz, H., Valdes, A., and Garvey, T. *A Real-time Intrusion Detection Expert System (IDES).* Tech. rep., SRI International, Computer Science Laboratory, February 1992.

[116] Lynch, N. *Distributed Algorithms.* Morgan Kaufmann Publishers. San Mateo, CA, 1996.

[117] Mandujano, S. *Toward the Preservation of a Key Feature of the Internet: Policy and Technology for Internet Anonymity.* International Conference on Technology Policy and Innovation, ICTPI '02, Monterrey, Mexico, 2002.

[118] Mandujano, S., and Galván, A. *ODISET: Online Distributed Session Tracing using Agents.* 18th International Joint Conference on Artificial Intelligence, IJCAI '03, pp. 749–754, Acapulco, Mexico, August 2003.

[119] Mandujano, S., and Galván, A. *Outbound Intrusion Detection.* International Conference on Computing, Communications and Control Technologies, CCCT '04, vol. 1, pp. 68–73, Austin, TX, August 2004.

[120] Mandujano, S., and Soto, R. *Deterring Password Sharing: User Authentication via Fuzzy c-Means applied to Keystroke Profiles.* 5th ENC International Conference in Computer Science, ed. IEEE Computer Society, pp. 181–187, Colima, Mexico, September 2004.

[121] Mann, D. E., and Christey, S. M. *Towards a Common Enumeration of Vulnerabilities.* 2nd Workshop on Research with Security Vulnerability Databases, Purdue University, West Lafayette, IN, January 1999.

[122] Marin, O., Sens, P., Briot, J. P., and Guessoum, Z. *Towards Adaptive Fault Tolerance for Distributed Multiagent Systems.* ERSADS 2001, pp. 195–201, Bertinoro, Italy, 2001.

[123] McAfee. *Entercept Security: Advanced e-Server Protection*, McAfee Security, Inc., February 2004. URL: http://www.nai.com/us/index.asp (on-line resource).

[124] McCarthy, L. *Symantec Internet Security Threat Report.* Tech. rep., Symantec, Inc., September 2003.

[125] McClure, S., Scambray, J., and Kurtz, G. *Hacking Exposed: Network Security Secrets & Solutions*, 4th ed. McGraw-Hill. February 2003.

[126] McHugh, J. *Intrusion and Intrusion Detection.* International Journal of Information Security, pp. 14–35, vol. 1, num. 1, 2001.

[127] Mell, P. *ICAT Metabase and CVE Vulnerability Search Engine*, Computer Security Division Information Technology Laboratory National Institute of Standards and Technology, NIST, 2000.

[128] Meunier, P., Mandujano, S., Song, W. F., and Ma, L. *Cooperativy Vulnerability Database.* Tech. rep., Center for Education and Research in Information Assurance and Security, CERIAS, Purdue University, West Lafayette, IN, February 2001.

[129] MimeStar. *SecureNet Pro*, MimeStar Intrusion Detection Co., March 2004. URL: http://www.mimestar.com/products/ (on-line resource).

[130] Mitchell, T. M. *Machine Learning.* McGraw Hill. 1st ed., 1997.

[131] Moitra, A. *Real-time Audit Log Viewer and Analyzer.* 4th Workshop on Computer Security Incident Handling, (Forum of Incident Response and Security Teams, FIRST), Denver, CO, August 1992.

[132] Moore, A. P., Ellison, R. J., and Linger, R. C. *Attack Modeling for Information Security and Survivability.* Tech. Rep. SEI-2001-TN-001, Survivable Systems, Carnegie Mellon University, Software Engineering Institute (SEI), Pittsburgh, PA, March 2001.

[133] Morini, P. *CERN Network Security Monitor.* 1st Workshop on Recent Advances in Intrusion Detection, Louvain la Neuve, Belgium, September 1998.

[134] Mouratidis, H., Giorgini, P., and Manson, G. *An Ontology for Modeling Security: The Tropos Approach.* Tech. rep., Department of Computer Science, University of Sheffield, Sheffield, UK, 2003.

[135] Mukkamala, S., Janowsky, G., and Sung, A. H. *Intrusion Detection using Neural Networks and Support Vector Machines.* IEEE International Joint Conference on Neural Networks 2002, pp. 1702–1707, Hawaii, May 2002.

[136] Mutz, D., Vigna, G., and Kemmerer, R. *An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems.* 19th Annual Computer Security Applications Conference, ACSAC, pp. 374–383, Las Vegas, Nevada, December 2003.

[137] Neto, H. V., and Borges, D. L. *Fingerprint classification with neural networks.* 4th Brazilian Symposium on Neural Networks, SBRN, IEEE CS Press, pp. 66–72, Campos do Jordao, Brazil, December 1997.

[138] NETWORK ASSOCIATES. *Cybercop Monitor*, Network Associates, Inc., February 2004. URL: http://www.tlic.com/security/cybercopmonitor.cfm (on-line resource).

[139] Network Solutions. *DefenseWorx*, Network Solutions, Inc., March 2004. URL: http://www.defenseworx.com (on-line resource).

[140] Neumann, P. G., and Porras, P. A. *Experience with EMERALD to Date.* 1st USENIX Workshop on Intrusion Detection and Network Monitoring, pp. 73–80, Santa Clara, CA, April 1999.

[141] NFR. *Centrax: Hybrid Security Solution*, NFR Security Corp., March 2001. URL: http://www.nfr.com/ (on-line resource).

[142] Northcutt, S., Cooper, M., Fearnow, M., and Frederick, K. *Intrusion Signatures and Analysis*, 1st ed. New Riders, SANS GIAC. Indianapolis, IN, January 2001.

[143] Northcutt, S., and Novak, J. *Network Intrusion Detection*, 3rd ed. New Riders. Indianapolis, IN, September 2002.

[144] Noy, N. F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. W., and Musen, M. A. *Creating Semantic Web Contents with Protege-2000.* IEEE Intelligent Systems, vol. 16(2), pp. 60–71, 2000.

[145] NSWC. *Shadow ID*, Naval Surface Warfare Center, December 2003. URL: http://www.nswc.navy.mil/ISSEC/CID/ (on-line resource).

[146] O'Hare, G. M. P., and Jennings, N. R. *Foundations of Distributed Artificial Intelligence.* John Wiley & Sons. New York, NY, 1996.

[147] Okazaki, Y., and Sato, I. *A New Intrusion Detection Method based on Process Profiling.* IEEE Symposium on Applications and the Internet (SAINT), pp. 82–91, Nara, Japan, January 2002.

[148] OKENA. *Stormwatch, white paper*, Okena Systems Co., October 2000. URL: http://www.okena.com/products/literature.htm (on-line resource).

[149] OSVDB. *Open Source Vulnerability Database*, May 2004. URL: http://www.osvdb.org (on-line resource).

[150] PacketStorm. *Packet Storm Security*, June 2003. URL: http://www.packetstormsecurity.org (on-line resource).

[151] Parunak, H. V. *An Overview of Distributed Artificial Intelligence.* Foundations of Distributed Artifical Intelligence, 6th Generation Computer Technology Series, G. OḤgare and N. Jennings, Eds., John Wiley and Sons, pp. 3–55, June 1996.

[152] Paxson, V. *Bro: A System for Detecting Network Intruders in Real-time.* Elsevier Computer Networks, pp. 2435–2463, vol. 31, num. 23–24, 1998.

[153] Pérez, J. A., Álvarez, D., and Sobrado, I. *A Fast Data Protection Technique for Mobile Agents against Malicious Hosts.* Electronic Notes in Theoretical Computer Science, K. Fischer and D. Hutter, Eds., vol. 63, Elsevier, 2002.

[154] Pescatore, J. *Nimda Worm Shows You Can't Patch Fast Enough*, Gartner Group, September 2001. URL: http://www.emailsolutions.com/gwnews/gartner_iis.pdf (on-line resource).

[155] PestPatrol. *Denial of Service Attacks on the Rise, whitepaper*, PestPatrol, Inc., July 2002. URL: http://www.pestpatrol.com/WhitePapers (on-line resource).

[156] Pfleeger, C. P. *Security in Computing*, 2nd ed. Prentice Hall, Inc. Upper Saddle River, NJ, 1997.

[157] Pleisch, S., and Schiper, A. *Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems.* 9th IEEE Symposium on Reliable Distributed Systems (SRDS 2000), IEEE Computer Society, pp. 11–20, Nurnberg, Germany, 2000.

[158] Polla, D., McConnell, J., Johnson, T., Marconi, J., Tobin, D., and Frincke, D. *A FrameWork for Cooperative Intrusion Detection.* 21st National Information Systems Security Conference, pp. 361–373, October 1998.

[159] Poole, D. L., MackWorth, A. L., and Goebel, R. G. *Computational Intelligence: A Logical Approach.* Oxford University Press. New York, NY, 1998.

[160] PRC. *PRC-Precis whitepaper*, PRC Systems, 1999. URL: http://www.bellevue.prc.com/precis/solution.pdf (on-line resource).

[161] Ptacek, T. H., and Newsham, T. N. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection.* Tech. rep., Secure Networks, Inc., January 1998.

[162] Quin, L. *Extensible Markup Language (XML).* Tech. rep., World Wide Web Consortium (W3C), 2002.

[163] Ranum, M. J. *Full Disclosure and Open Source*, Black Hat Briefings, keynote speech, 2000.

[164] Ranum, M. J., Landfield, K., Stolarchuk, M., Sienkiewicz, M., Lambeth, A., and Wall, E. *Implementing a Generalized Tool For Network Monitoring.* 11th Systems Administration Conference (LISA), USENIX, pp. 1–8, San Diego, CA, October 1997.

[165] Rapalus, P. *Computer Crime & Security Survey '02*, Computer Security Institute (CSI) and Federal Business of Investigations (FBI), April 2002.

[166] Raskin, V., Hempelmann, C. F., Triezenberg, K. E., and Nirenburg, S. *Ontology in Information Security: a Useful Theoretical Foundation and Methodological Tool*. New Security Paradigms Workshop, ACM Press, pp. 53–59, Cloudcroft, NM, 2001.

[167] Rathmell, A., and Valeri, L. *Information Warfare and the Asymmetric Threat: An approach to early warning*. Tech. rep., International Centre for Security Analysis, King's College London, London, UK, 1997.

[168] Reiter, M., and Rubin, A. *Anonymous Web Transactions with Crowds*. Communications of the ACM, pp. 32–38, vol. 42, num. 2, 1999.

[169] Richardson, R. *Computer Crime & Security Survey '03*, Computer Security Institute (CSI) and Federal Business of Investigations (FBI), April 2003.

[170] Roesch, M. *Snort: Lightweight Intrusion Detection for Networks*. 13th Conference on Systems Administration, pp. 229–238, Berkeley, CA, 1999.

[171] Ru, W. G., and Eloff, J. H. *Enhanced Password Authentication through Fuzzy Logic*. IEEE Expert, vol. 12–6, pp. 38–45, Nov/Dec 1997.

[172] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 1st ed. Prentice Hall. Upper Saddle River, NJ, January 1995.

[173] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall. Upper Saddle River, NJ, 2003.

[174] Ryan, J., Lin, M.-J., and Miikkulainen, R. *Intrusion Detection with Neural Networks*. Advances in Neural Information Processing Systems, M. I. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10, The MIT Press, pp. 943–949, Denver, CO, 1998.

[175] Samaha, S. E. *Haystack: An Intrusion Detection System*. 4th Aerospace Computer Security Applications Conference, vol. XII(440), IEEE Computer Society Press, pp. 37–44, Washington, DC, 1988.

[176] Sanctum. *AppShield: Automatic Web Application Control and Security*, Sanctum, Inc., January 2004. URL: http://www.sanctuminc.com/solutions/appshield/ (on-line resource).

[177] Sander, T., and Tschudin, C. F. *Protecting Mobile Agents Against Malicious Hosts*. Lecture Notes in Computer Science, LNCS, pp. 44–49, vol. 1419, num. 1, 1998.

[178] Schneider, F. B. *Towards Fault-tolerant and Secure Agentry*. 11th International Workshop on Distributed Algorithms, WDAG, M. Mavronicolas and P. Tsigas, Eds., Springer Verlag, Berlin, Germany, 1997.

[179] Schneier, B. *Applied Cryptography*, 2nd ed. John Wiley and Sons. New York, NY, 1996.

[180] Schneier, B. *Managed Security Monitoring: Network security for the 21st century.* Computers and Security, pp. 491–503, vol. 20, num. 6, 2001.

[181] Sebring, M. M., Shellhouse, E., Hanna, M. E., and Whitehurst, R. A. *Expert systems in intrusion detection: A case study.* National Institute of Standards and Technology/National Computer Security Center, 11th National Computer Security Conference, pp. 74–81, Baltimore, MD, October 1988.

[182] Sebring, M. M., Shellhouse, E., Hanna, M. E., and Whitehurst, R. A. *MIDAS: Multics Intrusion Detection and Alerting System.* Tech. rep., National Computer Security Center, SRI International, Ft. Meade, MA, 1988.

[183] Sebyala, A. A., Olukemi, T., and Sacks, L. *Active Platform Security through Intrusion Detection using Naive Bayesian Networks for Anomaly Detection.* London Communications Symposium, 2002.

[184] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. *Automatically Characterizing Large Scale Program Behavior.* ACM APLOS X, pp. 45–57, San Jose, CA, October 2002.

[185] Siaterlis, C., and Maglaris, B. *Towards Multisensor Data Fusion for DoS Detection.* ACM Symposium on Applied Computing, March 2004.

[186] Sirer, E. G., Renesse, R. V., Rodríguez, R., and Mandujano, S. *Dynamic Analysis of Executable Code using Function-call Graphs for Intrusion Detection.* Tech. rep., Department of Computer Science, Cornell University, Ithaca, NY, December 2003.

[187] Snapp, S. R., Smaha, S. E., Teal, D. M., and Grance, T. *The DIDS (Distributed Intrusion Detection System) Prototype.* Summer 1992 USENIX Conference, pp. 227–233, Oakland, CA, June 1992.

[188] Sobirey, M. *The Intrusion Detection System AID*, University of South Carolina, March 2004. URL: http://www.cse.sc.edu/research/isl/mirrorSobireys.shtml (on-line resource).

[189] Sobirey, M., Fischer-Hübner, S., and Rannenberg, K. *Pseudonymous Audit for Privacy Enhanced Intrusion Detection.* 13th International Information Security Conference, pp. 151–163, Copenhagen, Denmark, May 1997.

[190] Somayaji, A., and Forrest, S. *Automated Response Using System-Call Delays.* 9th USENIX Security Symposium, ACM Press, pp. 185–198, Denver, CO, August 2000.

[191] SourceFire. *OpenSnort*, Sunfire, Inc., February 2004. URL: http://www.sourcefire.com (on-line resource).

[192] Spafford, E. H., and Zamboni, D. *Data Collection Mechanisms for Intrusion Detection Systems.* Tech. rep., Center for Education and Research in Information Assurance and Security, CERIAS, Purdue University, West Lafayette, IN, June 2000.

[193] Spitzner, L. *Honeypots: Tracking Hackers*, 1st ed. Addison-Wesley Professional. September 2002.

[194] Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., and Zerkle, D. *GrIDS – A Graph-based Intrusion Detection System for Large Networks.* 19th National InfIntrusion Detection Working Group – ormation Systems Security Conference, vol. 1, pp. 361–370, October 1996.

[195] Staniford-Chen, S., and Heberlein, L. *Holding Intruders Accountable on the Internet.* IEEE Symposium on Security and Privacy, pp. 39–49, Oakland, CA, 1995.

[196] Staniford-Chen, S., Tung, B., and Schnackenberg, D. *The Common Intrusion Detection Framework (CIDF).* Information Survivability Workshop, Orlando, FL, October 1998.

[197] Symantec. *Intruder Alert*, Symantec, Inc., March 2004. URL: http://securityresp.symantec.com/IA.html/ (on-line resource).

[198] Symantec. *ManHunt*, Symantec, Inc., April 2004. URL: http://securityresp.symantec.com/MH.html (on-line resource).

[199] Symantec. *NetProwler*, Symantec, Inc., March 2004. URL: http://securityresp.symantec.com/NP.html (on-line resource).

[200] Teng, H. S., Chen, K., and Lu, S. C. *Security Audit Trail Analysis Using Inductively Generated Predictive Rules.* 6th Conference on Artificial Intelligence Applications, IEEE Service Center, pp. 24–29, Piscataway, NJ, March 1990.

[201] Tinnagonsutibout, C., and Watanapongse, P. *A Novel Approach to Process-based Intrusion Detection System using Read-sequence Finite State Automata with Inbound Byte Profiler.* Information and Computer Engineering Postgraduate Workshop, 2003.

[202] Undercoffer, J. L., Joshi, A., Finin, T., and Pinkston, J. *A Target-Centric Ontology for Intrusion Detection.* 18th International Joint Conference on Artificial Intelligence, IJCAI, pp. 749–754, Acapulco, Mexico, July 2003.

[203] Undercoffer, J. L., Joshi, A., Finin, T., and Pinkston, J. *A Target Centric Ontology for Intrusion Detection: Using DAML+OIL to Classify Intrusive Behaviors. Knowledge Engineering Review, Cambridge University Press*, 2004.

[204] Uschold, M., and Grüninger, M. *Ontologies: Principles, Methods, and Applications.* Knowledge Engineering Review, pp. 93–155, vol. 11, num. 2, 1996.

[205] Vaccaro, H. S., and Liepins, G. E. *Detection of Anomalous Computer Session Activity.* IEEE Computer Society Symposium on Security and Privacy, pp. 280–289, May 1988.

[206] VanderAalst, W., Aldred, L., Dumas, M., and TerHofstede, A. *Design and Implementation of the YAWL System*. Tech. rep., Department of Technology Management–Eindhoven University of Technology and Centre for IT Innovation–Quensland University of Technology, Queensland, Australia, 2004.

[207] Viega, J., and Messier, M. *Secure Programming Cookbook for C and C++*, 2nd ed. O'Reilly & Associates. July 2003.

[208] Vigna, G., and Jonnson, E., Eds. *6th International Symposium, Recent Advances in Intrusion Detection 2003* (September 2003), Lecture Notes in Computer Science, LNCS, Springer.

[209] Vigna, G., and Kemmerer, R. *NetSTAT: A Network-based Intrusion Detection System*. Journal of Computer Security, pp. 37–71, vol. 7, num. 1, 1999.

[210] Vytopil, J. *Formal Techniques in Real-time and Fault-tolerant Systems*. Springer-Verlag. Berlin, Germany, December 1991.

[211] Wagner, D., and Dean, D. *Intrusion Detection via Static Analysis*. IEEE Symposium on Security and Privacy, pp. 156–169, Washington, DC, 2001.

[212] Wang, L.-X. *A Course in Fuzzy Systems and Control*. Prentice Hall, Inc. Upper Saddle River, NJ, 1997.

[213] Wang, X., Reeves, D. S., and Wu, S. F. *Tracing-based Intrusion Response*. Journal of Information Warfare, vol. 1, num. 1, 2001.

[214] Weiss, G. *Multiagent Systems: A Modern Introduction to Distributed Artificial Intelligence*, 1st ed. MIT Press. Cambridge, MA, 1999.

[215] White, G. B., Fisch, E. A., and Pooch, U. W. *Cooperating Security Managers: A Peer-Based Intrusion Detection System*. IEEE Network, pp. 20–23, vol. 10, num. 1, 1996.

[216] Wimer, S. M. *Cylantsecure: A Scientific Approach to Security, whitepaper*, Cylant Corp., February 2004. URL: http://www.cylant.com/products/cylantsecure.htm (on-line resource).

[217] Wylie, J. J., Bigrigg, M. W., Strunk, J. D., Ganger, G. R., Kiliççöte, H., and Khosla, P. K. *Survivable Information Storage Systems*. Computer, pp. 60–68, vol. 33, num. 8, 2000.

[218] Yang, J., Ning, P., Wang, X. S., and Jajodia, S. *CARDS: A Distributed System for Detecting Coordinated Attacks*. IFIP TC11 Sixteenth Annual Working Conference on Information Security (SEC 2000), pp. 171–180, August 2000.

[219] Yao, Y., Frasconi, P., and Pontil, M. *Fingerprint Classification with Combinations of Support Vector Machines*. Lecture Notes in Computer Science, LNCS, pp. 253–258, vol. 2091, num. 1, 2001.

[220] Ye, N., Li, X., Chen, Q., Emran, S. M., and Xu, M. *Probabilistic Techniques for Intrusion Detection Based on Computer Audit Data.* IEEE Transactions on Systems, Mand, and Cybernetics, vol. 31(4), pp. 266–274, July 2001.

[221] Yee, B. S. *A Sanctuary for Mobile Agents.* Secure Internet Programming: Security Issues for Distributed and Mobile Objects, Springer Verlag, Lecture Notes in Computer Science, LNCS, pp. 261–273, Berlin, Germany, 1999.

[222] Yoda, K., and Etoh, H. *Finding a Connection Chain for Tracing Intruders.* 6th European Symposium on Research in Computer Security, ESORICS, pp. 191–205, October 2000.

[223] Zamboni, D. *Doing Intrusion Detection using Embedded Sensors.* PhD thesis, Purdue University. West Lafayette, IN, 2000.

[224] Zamboni, D. M., and Spafford, E. H. *Intrusion Detection using Autonomous Agents.* Computer Networks – Elsevier, pp. 547–570, vol. 34, num. 04, 2000.

[225] Zou, C., Gong, W., and Towsley, D. *Code Red Worm Propagation Modeling and Analysis.* 9th ACM Conference on Computer and Communications Security, pp. 138–147, Washington, DC, 2002.